UNIVERSITY OF SOUTHAMPTON

FACULTY OF PHYSICAL AND APPLIED SCIENCES

Electronics and Computer Science

Formal Engineering Methodologies for Wireless Sensor Network Development with Simulation

by

Adisak Intana

Thesis for the degree of Doctor of Philosophy

December 2015

UNIVERSITY OF SOUTHAMPTON

ABSTRACT

FACULTY OF PHYSICAL AND APPLIED SCIENCES
Electronics and Computer Science

Doctor of Philosophy

FORMAL ENGINEERING METHODOLOGIES FOR WIRELESS SENSOR NETWORK DEVELOPMENT WITH SIMULATION

by Adisak Intana

In the current practice in Wireless Sensor Network (WSN) development, the software code representing communication protocols and algorithms tends to be complex, monolithic, and close to low-level operating system and hardware platforms. The software is thus difficult to understand and maintain. This is because of the lack of software engineering (SE) and model-based techniques, tools and infrastructure in the WSN domain. Furthermore, critical design requirements must be guaranteed, as uncertain and unreliable environment circumstances may cause the failure of a WSN deployment, e.g. by node death and communication failures. Therefore, good SE methodologies and techniques including high-level abstraction, separation of concerns, powerful verication and validation (V&V) are needed for WSN development.

This thesis proposes a Formal Co-simulation (*FoCoSim-WSN*) framework to strengthen current SE practice for WSN development. This framework enables an iterative and incremental development style which combines the benefits of existing simulation and proof-based formal verification approaches and tools. The complexity of software development for the sensor node controller is reduced by separating the controller model from the simulation environment. Controller algorithms for each protocol stack layer can be formally developed and veried in a layered manner using the refinement method of the Event-B language and its RODIN toolkit. The absence of certain classes of faults in controller models which cannot be guaranteed by simulation testing techniques, can be proved by formal methods. The MiXiM simulation of physical environment provides confidence in the reliability and performance analysis through long running simulation via wireless channels.

Our prototype development confirms the flexibility, usability and reusability of the framework for interworking between formal, simulation and co-simulation modelling. Furthermore, the integration of our proposed FoCoSim-WSN framework with the Model-Based Trace Testing (MBTT) approach gives us confidence in the validation coverage. Test scenarios including functional, failing and recovery tests are created from the sequence of events in our co-simulation master algorithm. Long-running test scenarios generated by MiXiM co-simulation enable model debugging for absent or erroneous constraints and events in our formal controller. Finally, by investigating two case studies we identify reuse opportunities and propose reusable patterns for Event-B and master models.

Contents

List of Figures

List of Tables

Declaration of Authorship

I, Adisak Intana , declare that the thesis entitled *Formal Engineering Methodologies for Wireless Sensor Network Development with Simulation* and the work presented in the thesis are both my own, and have been generated by me as the result of my own original research. I confirm that:

- this work was done wholly or mainly while in candidature for a research degree at this University;
- where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated;
- where I have consulted the published work of others, this is always clearly attributed;
- where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work;
- I have acknowledged all main sources of help;
- where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself;
- parts of this work have been published as: (Intana et al., 2013a), (Intana et al., 2013b), (Intana et al., 2014) and (Intana et al., 2015)

Signed:.....

Date:.....

Acknowledgements

This thesis would not have been possible without the guidance of my supervisors and the help and support from my colleagues, friends and family.

Firstly, I am truly indebted and thankful to my supervisor, Dr. Michael Poppleton, not just for his excellent guidance and advice but also for his support and encouragement in carrying out my PhD research and thesis. I would also like to express my gratitude to my adviser, Dr. Geoff Merrett, who supported me, especially the guidance and discussion about wireless sensor network issues.

Secondly, I would wish to owe my deepest gratitude to Gerry Cox who helped me proof-read this thesis and supported and encouraged me throughout the time of my study.

Furthermore, it is a great pleasure to thank the Royal Thai Government for granting me a scholarship to complete my PhD.

I would also like to thank my family in Thailand, particularly my parents, grandparents and sisters, for their love, blessing and support. They have always contributed to my emotional well-being, especially in hard times, throughout the completion of my PhD.

Lastly, I am obliged to many of my colleagues and friends in Electronic and Software Systems (ESS), University of Southampton. I would like to deliver special thanks to Dr. Tristan Aubrey-Jones, Dr. Sanaz Yeganefard and Tim Thomas, Dr. Somjit Sriya, Suthira Plansangket, Boonyarat Phadermrod, Dr. Woraluck Wongse-ek, Dr. Nakarin Pinpathomrat and Dr. Jarutas Pattanaphanchai for their great friendship and company that provided me emotional well-being during my PhD study.

Nomenclature

ACK ..... Acknowledgement
CS ..... Carrier sense
CTS ..... Clear-To-Send ETDEvent trace diagram
FoCoSim-WSN ..... Formal Co-simulation framework for WSNs
MAC ..... Medium Access Control
MBTT ..... Model-Based Trace Testing
MDE ..... Model-Driven Engineering
MUT ..... Model Under Test
NAV ..... Network Allocation Vector
OS ..... Operating System
PO ..... Proof Obligation
RTMCS ..... Road Tunnel Monitoring and Control System
RTS ..... Request-To-Send
SE ..... Software Engineering
S-MAC ..... Sensor MAC
V&V ..... Verification and Validation
WSN ..... Wireless Sensor Network
SINR ..... Signal-to-Interference-plus-Noise Ratio
SUT ..... System Under Test
SYN ..... Synchronisation

Chapter 1

Introduction

1.1 Research Challenges for Current Practice in WSN development

A Wireless Sensor Network (WSN) is a distributed system of cooperating embedded devices called sensor nodes which are deployed to monitor physical real-world phenomena over a self-organised wireless network topology. Advances in low-power micro-embedded devices together with short-range wireless communications means that WSN is increasingly being adopted within a number of scientific fields particularly safety-critical domains such as healthcare monitoring (Baldus et al., 2004) and volcano monitoring sensor system (Werner-Allen et al., 2006; Challen and Welsh, 2010). Despite the powerful capabilities of this technology, the successful development and deployment of WSN is still a challenging task. In current practice in WSN development, programming needs to be very close to simple WSN operating systems (OSs) (Picco, 2010; Farooq and Kunz, 2011; Mottola and Picco, 2011, 2012; Doddapaneni et al., 2012). As these OSs are low-level systems providing a set of library packages connected to specific hardware, there is a weak level of software engineering (SE) supports for these OSs. This means that programmers are expected to produce their software code at lower-level in which software code is tied up with all required services and their lower-level APIs provided by OSs without the high-level of SE supports (Picco, 2010; Farooq and Kunz, 2011; Mottola and Picco, 2011, 2012; Doddapaneni et al., 2012). The result, however, is that the software code is difficult to understand and modify because of the complexity of low-level programming. In some cases, it is also unreliable as faults are discovered after releasing software code (Beutel et al., 2010) and there is a high rate of requirement change after deployment (Allen et al., 2010). Consequently, this causes highly coupled software code to be very difficult to maintain. Although middleware (Henricksen and Robinson, 2006; Chatzigiannakis et al., 2007; Mottola and Picco, 2012) has been introduced to reduce the complexity of this monolithic code, low-level system issues are still involved to be

in the focus of development. As a result of this, the design of WSN is still difficult as it need a much higher level of abstraction. Consequently, good software engineering (SE) methodologies and techniques are needed for WSN development.

1.1.1 Simulation-based Engineering Approaches

Recently, various SE methodologies and techniques have been introduced to improve the current practice of WSN development. One of the most widely adopted techniques is the simulation-based approach. This technique is generally applied at an early stage of designing and testing communication protocols because it provides the higher level of abstraction necessary (Kiess and Mauve, 2007; Imran et al., 2010) It abstracts away from specific operating system platforms whereas other similar techniques such as emulation and laboratory testbeds do not. In current simulation practice, protocols and algorithms are layered to create a communication networking protocol suite by a standard protocol stack scheme. These are integrated with a stochastic environment framework of wireless channel, radio and analogue models to generate long running testing scenarios. The simulation and performance analysis such as network latency and energy consumption are conducted independently of any specific platform (Köpke et al., 2008; Korkalainen et al., 2009; Sundani et al., 2011). However, code for simulation is developed monolithically. At each layer of the protocol stack, all functionality of a protocol algorithm has to be integrated with simulation environments provided by standard interface in simulation toolkit e.g. channel communication and connectivity and the library function - implementing packet transmission mechanism in a single simulation model. Consequently, current practice is a long way from the model-based software engineering process. The specifications of the behaviours of the software controller algorithm and the behaviour of the environment are implemented in simulation at the same time (Köpke et al., 2008; Boulis, 2011; Fall, 2011). This means that significant complexity needs to be managed during development, and this makes it hard to understand the code. Thus, a clear separation of concerns is required.

As higher-level abstractions and separation of concerns are needed by WSN development, several studies propose the integration of model-driven engineering (MDE) techniques and technologies with simulation framework to strengthen the current WSN development (Shimizu et al., 2012; Wang and Baras, 2012; Doddapaneni et al., 2012). MDE creates multiple abstraction levels for WSN developments in which software components containing computation and communication algorithms, low-level hardware specification of nodes and the physical environment are modelled separately. An abstract model is described by using a Domain Specific Modelling Language (DSML) e.g. Systems Modelling Language (SysML)[1] (OMG, 2012), customizing UML (Unified Modeling Language) for a specific domain. Then, model transformation is used to transform this abstract model

[1] see SysML -http://www.omg.org/spec/SysML/1.3/

into a more concrete (executable) model simulated in the simulation. However, these integration approaches do not include highly reliable verification and validation (V&V) techniques which are needed for the development of WSNs in safety-critical domains. Furthermore, it is not guaranteed that critical design errors will be discovered during simulation:- these techniques cannot guarantee the absence of certain classes of faults as formal methods do (Hall, 1990; Dijkstra, 1969).

1.1.2 Formal-based Engineering Approaches

In the last few decades, the use of formal methods (FMs) for network protocol specification and verification has been widespread (Cardell-Oliver, 1991; Rushby, 1997; Bhargavan et al., 2002; Abrial et al., 2003; Méry and Singh, 2011a). In the domain of WSNs, FMs have been adopted into this domain as FMs can serve the needs of high and reliable V&V methodologies and techniques demanded by WSNs in safety-critical domains. By adopting FMs, the influential benefits are derived in which the absence of certain safety standards at an early stage of development can be guaranteed as mentioned above. Furthermore, FMs offer clear specification at high-level abstraction and powerful verification techniques to verify all possible interleaving is free from error. Therefore, FMs are becoming widely recognised as a solution to manage the complexity of the development and reduce potential failures of WSNs before the deployment (Nair and Cardell-Oliver, 2004; Ölveczky and Thorvaldsen, 2007; Killian et al., 2007; Saghar et al., 2010).

Despite this applicability of FMs, the practical development of formal models is very limited. Most formal developments simplify the functionality under the simplified assumption. For example, in the proof-based formal verification, the use of theorem-proving tools is beneficial in exploring model properties in order to detect the conflict between different requirements and missing assumptions. However, this results in the only verified formal specification which lacks realistic behaviours. Basically, the formal model is verified and validated under the simplified environment assumption where verification scenarios are generated. Although the correctness of protocol algorithms is proved successfully by the theorem prover, but this does not translate a guarantee that the implementation is error-free as some verification scenarios regarding realism may not be generated by this very abstract environment. Furthermore, the complexity and scalability of WSN applications need long running simulation behaviours to ensure full confidence about reliability and performance requirements in formal models. Consequently, the executable formal method is required by the network protocol development.

Therefore, several research studies have proposed a hybrid approach between formal methods and simulation testing in order to provide an executable formal method under more concrete environment for WSNs (Nair and Cardell-Oliver, 2004; Samper et al.,

2006; Wang et al., 2009; Matousek et al., 2010; Niazi and Hussain, 2011). Their contribution is to strengthen the current practice of development which helps WSN engineers improve their confidence in the correctness of their executable specification model together with its performance evaluation in different environments. However, to our knowledge, most of these work studies only proposed the specification model which can be executed by simulation:- they did not show any proof to guarantee how all desired properties always hold. Furthermore, most research studies abstract their environment models away from the real environment in order to be executed with their formal models. The concrete environment which is very close to the real environment is needed by this approach. For example, FABS (a formal agent-based simulation) framework (Niazi and Hussain, 2011) enables the physical environment, the sensor node and the sensor environment to be modelled formally by using Z (Spivey, 1989; Bowen, 2003) formal language. These formal specification models can be translated into the system simulation model and driven by NetLogo (Wilensky, 1999), an agent-based simulation tool. However, we have found two limitations in this work. Firstly, they did not show any proof in their formal model. Secondly, their developed environment model lacks realism. For instance, this environment encoded only the simple equation for calculating location of each sensor network that is a long way from realistic environment e.g. the maximal interference distance calculation to initiate the connection between two nodes. Therefore, the goal of this research is a proposed novel approach which enables the integrated framework for co-simulating between the executable formal controller models and more concrete and accurate environment models provided by WSN simulation.

1.1.3 Research Challenges

Based on the research opportunity we identified from the current practice in WSN development as mentioned in the earlier sections, this research proposes the engineering approach integrating proof-based formal and testing-based simulation technologies to increase the quality of current SE practice for WSN development, including high-level abstraction, separation of concerns, and powerful V&V techniques. We construct the infrastructure for co-simulation between formal Event-B WSN models and MiXiM environment simulation engines. We describe our research vision in Section 1.2, before presenting our research contribution in Section 1.3. Finally, in Section 1.4, we discuss our thesis structure.

1.2 Research Vision

Figure 1.1 shows our research vision for our co-simulation framework. This provides an integrated set of methodologies for WSNs: (S)imulation, (F)ormal and (C)o-simulation.

The *FoCoSim-WSN* framework is proposed based on this vision which is a formal co-simulation method for WSNs that integrates between the formal Event-B modelling and verification technique and MiXiM simulation testing technologies.

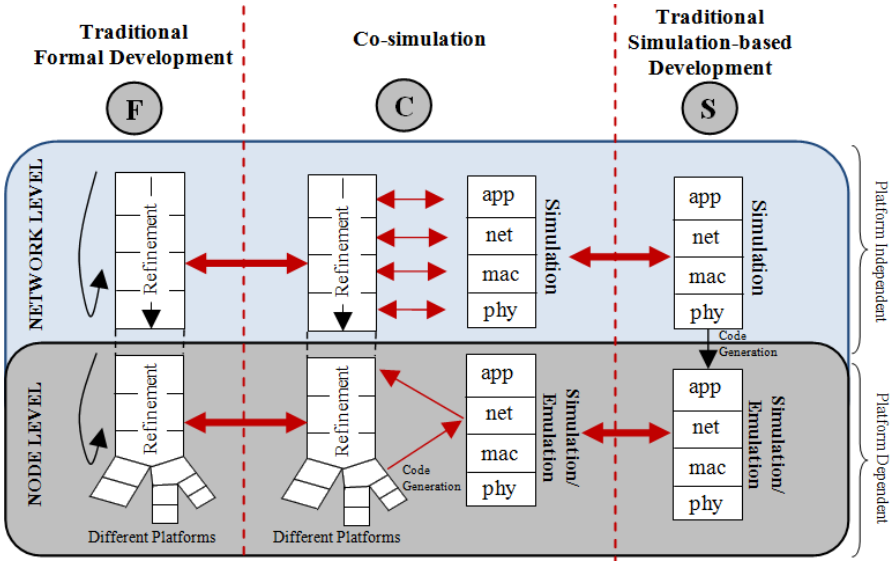


Figure 1.1: Vision of co-simulation approach for WSN development

(S) S-style development is the traditional WSN development style that layers the protocol algorithms and evaluates the network performance (Kiess and Mauve, 2007; Köpke et al., 2008; Imran et al., 2010) as mentioned earlier. Target code based on the simulation model is generated making use of standard platform-specific libraries. Node-level simulation or emulation takes place to test the correctness and performance of the real code before its real-world deployment.

(F) F-style modelling represents the requirement specification, modelling and verification in a formal modelling language. Each protocol algorithm is layered and verified through design and proof of refinement steps at network-level development. The verified network model is produced before different refinement paths are encoded with requirements for the specific dependent platform at node level. The final, verified node code is generated with standard libraries for a specific hardware platform from this verified node model.

(C) Our C-style prototype framework reduces the complexity of development by separating the software controller from the environment. Formal methods provide the controller, a formal model of code in the real nodes containing the protocol algorithms separately. An environment simulator provides stochastic sensed data and radio environment, allowing simulation scenarios to be defined as required. The verified controller model for each layer of protocol stack - ultimately, down to verified generated code is co-simulated with the environment model to perform the performance analysis. A master co-simulation language and algorithm is required

to integrate and manage the component simulators. The formal Event-B controller model simulated by ProB can co-simulate with a sensor environment provided by MiXiM via this master.

1.3 Research Contribution

Our main research contribution is a formal co-simulation approach (*FoCoSim-WSN*) to increase the quality of WSN development. The created co-simulation framework will help WSN engineers to improve their development processes by providing a flexible and reusable development environment. Furthermore, this framework will give the engineers confidence in powerful V&V and performance of the developed WSN application. Therefore, our main contributions can be divided into five sub-contributions:

C.1 The integration between proof-based formal verification and simulation-based testing techniques. This supports the encoding of the functionality and protocol algorithms into the controller model in the formal Event-B language. These techniques focus on high-level abstraction and separation of concerns. The refinement technique can manage the complexity of the development by layering the abstraction of the models into multiple concrete levels, whereas the decomposition technique decomposes the controller model representing protocol algorithms separately from the wireless and physical environment. Use of proof tools can guarantee the safety properties of this formal model (Chapter 6). Furthermore, network reliability and performance analysis are performed by MiXiM simulation including, for example, the network load distribution and the network latency, through long-running testing scenarios (Chapter 5 and 7).

C.2 Methodological flexibility in co-modelling development.

As each layer protocol development may be implemented by different engineers, our developed FoCoSim-WSN framework provides an iterative and incremental interworking scheme through multiple refinement levels. This results in the multiple levels of co-simulation abstraction that distributes the functionality of WSN systems including protocol algorithms between components. This gives flexibility in WSN development to integrate between F-, S- and C-style modelling. In order to demonstrate this, two strategies of interworking evolution are proposed: *from-S-through-F-to-C*(S^F *Co-simulation*) (Chapter 7) and *from-F-through-S-to-C* (F^S *Co-simulation*) (Chapter 9). These enable engineers to work either F- or S-style development before the separated controller and environment are combined and co-simulated into our prototype framework. System engineers can implement their models to easily cross into these three flexible modes of working.

C.3 Formal co-simulation interfaces contributing to reusability. Novel independent interfaces are designed and implemented under our proposed co-simulation framework. These interfaces are integrated with a socket API which enables flexible and sufficient co-simulation communication between the different platforms of formal Event-B controller and MiXiM simulation environment models. Furthermore, input/output parameters exchanged between these co-models are encoded in XML standard (Chapter 7).

C.4 Model-based trace testing (MBTT) technique. This technique benefits the validation coverage in the formal Event -B controller in co-models; it enables us to test scenarios to be created from the sequence of events in our co-simulation master algorithm. We use event-trace diagrams, fault injection and recovery testing to specify functional, failing and recovery test scenarios. We define MiXiM co-simulation runs to generate long-running test scenarios that meet our test requirements (Chapter 8).

C.5 Usability and Reusability in co-model development. In WSN development, engineers need to design specific operations, e.g. sensing and actuating for application, and routing calculation for routing protocols together with generic operations, e.g. communication mechanism between nodes. These generic operations have potential for reuse as patterns. Therefore, reusable refinement, master and simulation module patterns are provided in our FoCoSim-WSN framework (Chapter 9).

1.4 Thesis Structure

This thesis is structured as follows:

Chapter 2

This chapter presents an overview of formal methods including the definition and advantages. This is also discussed together with the impact of formal engineering in industry. Furthermore, Event-B, which is a proof-based formal modelling language used in our research, is also explained in this chapter.

Chapter 3

This chapter sets out the relevant background of wireless sensor networks (WSNs). The limitation of current SE practices in WSN development including FMs for WSNs is also explored in this chapter in order to identify research challenges. Furthermore, as this research contributes the benefit of using formal-based development to simulation-based approaches for WSNs, an overview of WSN simulators with their benefits and drawbacks is given in this chapter. Furthermore, WSN deployment problems which we inject into our case study are also explored.

Chapter 4

This chapter presents the case study, *SensorScope*, an environment monitor system, to demonstrate our contributions. This case study is used to demonstrate the S-, F- and C-style development that are described in Chapter 5, 6 and 7 respectively.

Chapter 5

This chapter demonstrates the strengths of S-style development in which the performance of protocol algorithms, e.g. load distribution of the network and network latency, can be analysed and evaluated. Furthermore, the weaknesses of this development style are also explored to illustrate why this style of development needs to be integrated with formal methods. The practical building case study created in Chapter 4 is used to demonstrate these weaknesses (Sub-contribution C.1).

Chapter 6

This chapter discusses the F-style approach, showing the benefits of specifying and verifying the network algorithm model to strengthen the weaknesses of simulation discovered in Chapter 5. The refinement technique is used to reduce the complexity of development by layering the communication and protocol algorithms in a specific manner. Furthermore, we apply a shared-event decomposition to separate the controller from the environment. F-style co-simulation is introduced to co-simulate between the separated formal controller and environment in order to validate the controller model before deploying it into co-simulation framework proposed in Chapter 7 (Sub-contribution C.1).

Chapter 7

This chapter introduces our prototype C-style framework (*FoCoSim-WSN*) for co-simulation. This chapter also demonstrates how the co-simulation framework influences multiple abstraction levels for co-simulation through refinement. The flexibility in protocol algorithm development of this framework is demonstrated with the SensorScope case study (Sub-contributions C.1, C.2 and C.3).

Chapter 8

This chapter examines the validation of formal controller models in WSN co-simulation by applying model-based trace-testing technique. This technique can derive test scenarios from a formal Event-B model, a model under test (MUT). Different testing experiments are created from these test scenarios and configured in the configuration file for MiXiM, a test driver in the co-simulation framework. This chapter shows "failure test cases" ("killer traces") containing failure scenarios can influence the model debugging to reveal the absent or erroneous constraints and events (Sub-contribution C.4).

Chapter 9

This chapter describes the evaluation of our developed co-simulation (C-style) framework in terms of potential usability and reusability. To accomplish this, another case study, Road Tunnel Monitoring and Control System (RTMCS), is considered to demonstrate

the wider applicability and reusability of our framework. Three categories of reusability enabled by this framework are also discussed, including *reusability of formal controller*, *master implementation* and *simulation implementation*. Finally, this chapter proposes the refinement and generic Event-B patterns. We also demonstrate how our generic Event-B patterns can create the common reusable instantiation which provides directly reusable skeleton for the potential development (Sub-contribution C.5).

Chapter 10

Finally, this chapter provides a summary of this thesis. We discuss the conclusions that can be drawn from the results of this frameworks engineering process and V&V from our FoCoSim-WSN framework. Furthermore, we discuss future extensions to this work.

Chapter 2

Formal Method Background and Survey

2.1 Introduction

This chapter presents formal methods, techniques and tools used for specification and verification. To support our formal co-simulation framework engineering process, Section 2.2 discusses the reasons why software engineering considers formal methods. Next, the advantages of applying formal methods are explained in Section 2.3. In Section 2.4, we discuss the issues affecting the industrial adoption of formal methods before giving an overview of refinement in Section 2.5. Section 2.6 describes the classification of formal methods before some existing successful formal modelling languages are discussed in Section 2.7. In Section 2.8, Event-B which is used in our research is also discussed. Section 2.9 describes decomposition and refinement atomicity in Event-B. V&V techniques which are model checking and model animation are briefly discussed in Sections 2.10 and 2.11 respectively. This also includes the model-based testing technique discussed in Section 2.12. Section 2.13 provides the conclusion of this chapter.

2.2 Why Does Software Engineering Consider Formal Methods?

Formal methods have emerged within software engineering (SE) since the "software crisis" was discovered in the late 1960s (Gibbs, 1994). The term "software crisis" originated in late 1968 from the NATO Conference on Software Engineering (Naur and Randell, 1969) that addressed the major development issues resulting in a list of identified software problems. These included the unreliable nature of software; late delivery;

prohibitive in terms of modification costs; impossible to maintain; performing at an inadequate level, and exceeding budget costs. In the 1980s, many software engineering researchers proposed the formal development method for software quality improvement (Jones, 1990; Gibbs, 1994; Spivey, 1989; Sommerville, 2011). They argued that the use of formal methods which essentially focuses on the specification, analysis, design and verification would result in fewer program errors and achieve more user requirement satisfaction.

Formal methods are mathematically-based techniques which use mathematical notations to describe and give reasons for the property and behaviours of hardware and software systems. The capability of these mathematical foundations strengthens the weakness of classical software engineering by focusing on the specification and verification of the system.

In requirement engineering, requirement specifications are normally expressed through the use of informal languages, e.g. natural languages. Thus, this leads to many defects in requirement specifications. This includes defining unclear, ambiguous, hard to understand, incomplete and inconsistent requirements (Kandt, 2003; Firesmith, 2007). However, most formal specification languages use mathematical logic and set theory which means that the syntax and semantics of the model are formally defined. By using formal methods, a precise statement is developed describing the expected services or properties provided by software and the constraints system that must operate. Consequently, this enables the properties of the future system to be clearly understood, which helps software engineers to eliminate ambiguities and incompleteness of the system (Crow et al., 1995). Furthermore, formal methods establish the traceability checking between requirements and formal specification. This enables software engineers to be able to identify the incompleteness in system requirements and design as reported in Jastram et al. (2010); Alkhammash et al. (2013).

Furthermore, due to the software crisis particularly in the safety-critical domains, as discussed earlier, powerful V&V are needed by software engineering (Gibbs, 1994). In classical SE, the software testing technique is not sufficient to guarantee that the software product is free from error. This technique can only detect the presence of faults but it cannot guarantee the absence of certain classes of fault as formal methods like proof-based formal methods can do (Hall, 1990; Dijkstra, 1969). Formal methods include the generation of verification conditions which capture requirements and the ability to prove those by using proof. This affects the model of the system to be verified whether it satisfies all desired properties (Abrial, 2007). This proof mechanism supports the software engineer in the process of reasoning their model. For example, in state-based formal methods like Event-B, the properties of the system as described by requirements are expressed by invariants. These invariant properties must always hold. In order to accomplish this, every event must be proved to maintain the invariants. This creates the benefit that the program the software engineers are going to develop will be consistent

and the program errors can be detected in the early stages of the development. This leads to one of the benefits that of the likelihood of design errors occurring during development is considerably reduced and failure risks which may result in extremely expensive costs in the testing phase are eliminated.

2.3 Advantages of Formal Methods

Addressing the reason for why SE considers formal methods, the advantages of these methods are summarised below:

- The use of mathematical concepts resulting in a precise statement reduces defects in the specification which helps to develop a clear understanding of specifications, and assists in the elimination of ambiguities and incompleteness in the system.
- Existing experimental work of formal methods demonstrates higher rate of defect discovery at early life cycle phase during specification and modelling (Houston and King, 1991; Hall, 1999; Gerhart et al., 1994). Therefore, this benefits the cost of late fault discovery to be avoided.
- System analysts' and designers' confidence about the correctness of their developed system is improved when the absence of certain classes of faults is guaranteed by using formal verification. Traditional testing techniques cannot do this.

2.4 The Industrial Impact of Using Formal Methods

The successful adoption of ideal formal methods into practical industries has been aimed for over several decades. The main objective of this application is to reduce the defects at earlier stages of software development, e.g. specification and design, under real industrial conditions at reasonable cost. As the contribution of this thesis is to propose the formal co-simulation approach for the WSN engineering process, a literature survey regarding the impact of applying formal methods to the software industry has been undertaken and is described in this section.

By the 1990s, several studies had been undertaken showing that formal methods can actually improve the quality and the productivity of industrial software development processes. For example, Houston and King (1991) reported experiences and results from using Z within IBM's Customer Information Control System (CICS) project. The result shows that this application benefited the improvement of the product quality in the development process including earlier detection of errors discovered in the development process, a lower defect rate in the final product and an estimated 9% reduction in the total development cost. Other evidence is shown in the surveys of Craigen et al. (1993);

Hinchey and Bowen (1995):- they concluded that the use of formal methods is increasing in the development of critical systems, which is needed to guarantee the high degrees of safety, reliability and security required. Examples of popular adopted application domains are railway systems and air traffic control systems.

One of the successful adoptions of formal methods within industry is delivered by Praxis, the CCF Display Information System, one component of the new air traffic management system for London's airspace (Hall, 1999). Praxis integrated the formal method influenced by VDM (Vienna Development method) (Jones, 1990) to be become part of the development process, and co-joined this mathematical foundation with other software engineering techniques including project management and quality assurance. At the end of development, they found that the defect rate per thousand lines of code was 0.75 faults. This figure was two to ten times better than that of comparable software in air traffic control applications implemented by conventional software development (using informal methods).

The use of formal methods to implement railway systems emerged around 1998 (Bloomfield et al., 2000). Due to the success of this adoption, formal methods have been increasingly applied to a number of train protection products such as the B method and the Atelier B toolset for train application development of the Cairo and Calcutta Metros and Paris metro line (Behm et al., 1999). By applying formal methods, the costs of development were decreased by two orders of magnitude from the early developed projects as reported in Gerhart et al. (1994).

Despite this successful adoption, however, some researchers have discovered challenges leading to the failure of application of formal methods to practical software development in industries. Most of these challenges include the need for training in well-defined mathematical notations in formal specifications (Knight et al., 1997; Schulte, 2000; Bloomfield et al., 2000). Furthermore, they believe that adequate tool support is another challenge to encourage software industries to adopt the formal methods. The studies of Bloomfield et al. (2000) and Sommerville (2011) report that existing formal method tools are only concerned with small-scale system development. Consequently, in order to achieve tool support for complex and large-scale system development, integration of formal method techniques with other software engineering practices needs to be improved (Craigen et al., 1993). One of the trends that has been proposed is the formal co-simulation approach which integrates discrete-event models of formal controllers with continuous-time models of their simulation environments (Fitzgerald et al., 2013).

Based on the literature survey about the adoption of formal methods within the software industry, we discover that there is common agreement over which formal methods have been successfully adopted within industrial software development. Efficient tool support for formal software development is still required. Consequently, as current practice of SE for WSN domains lacks the high verification and validation techniques (Picco, 2010),

we undertake our research contribution by applying the formal engineering approach to contribute to effective engineering for WSN development. The formal co-simulation framework is developed which integrates the formal modelling and proof toolkit, Rodin (Abrial et al., 2006; Butler, 2007) with its capability executing formal models (Bendiposto et al., 2012; ADVANCE, 2013; Bendiposto et al., 2013), in Event-B together with the existing well-known WSN simulation engine, MiXiM (Köpke et al., 2008). This provides the flexible iterative for an interworking mode of development between formal, simulation and co-simulation modelling.

2.5 Refinement

Refinement has been introduced as a model-based specification methodology for synthesising a program from an abstract specification (DeRoever and Engelhardt, 1999). This structuring methodology is provided by many formal languages, such as ASM, B, Event-B, Z, VDM, and TLA+ among others in order to reduce complexity of development and bridge a developed specification towards a suitable implementation. Refinement enables the separation of concerns in development in the SE principle which builds a precise model in layers by separating requirement and design concerns. By using this approach, a very simple abstract model is implemented first which describes the overall goal of the system. Then, it stepwise specifies the system requirement and functionality in more detail, in concrete refinement. The structure of the model is also enriched by considering the system architecture including the data structure and algorithm in later refinements, in order to bring the model closer to the implementation level.

Using refinement, the difficulties of developing a large and complex system that contains a significant number of discrete transition components can be reduced. It helps software engineers to divide their modelling task into steps. Software engineers can start constructing simple abstractions of key requirements and then later create more concrete models with more detailed concrete requirements. Furthermore, while involved in each step of these refinements, software engineers can trace whether the definition and requirement(s) contained in the model are incomplete or inconsistent. As a result, software engineers can also discover any remaining definitions or requirements that have not been considered in their model. The final model is produced when there are no remaining definitions or requirements that have not been considered in this model.

2.6 Classification of Formal Methods

Based on the study of Liu et al. (1997b,a), formal methods are divided into five categories:

- **Model-based Approach:** a set-theoritic model of a system is built. In this model, states and operations which express the state transitions of the system are defined. Some examples of this category are Z (Spivey, 1989), VDM (Jones, 1990), B-Method (Abrial, 1996), Event-B (Abrial, 2007), and PVS (Owre et al., 1992).

- **Logic-based Approach:** desired system properties are defined by using logics. These logic properties are validated axiomatically. Hoare Logic (Hoare, 1969) and Temporal Logic (Manna and Pnueli, 1992) are two examples of this category.

- **Algebraic Approach:** the system is formally expressed by the types of data and mathematical operations on those data types. The behaviour of operations and transitions is defined by specifying the type of input data as a pre-condition and the type of output result as a post-condition on those operations. In this approach, the concurrency can be explicitly represented. Some examples of this category are OBJ (Goguen and Tardo, 1979) and LARCH (Guttag and Horning, 1993).

- **Process Algebra Approach:** this approach models and analyses the behaviour of concurrent discrete event systems by observing the communication between processes. One example of this formal approach is CSP (Hoare, 1978).

- **Net-Based Approach:** this approach uses the graphical notations with a formal definition and semantics to specify system behaviours/state transitions. Petri Net (Reisig, 1985) is one example of this approach.

2.7 Overview of Some Formal Modelling Languages

The formal language used in this research is Event-B, a model-based modelling language. This section provides an overview of some of other formal modelling languages which are Z, VDM, B and PVS. As Event-B is inspired by action system, an overview of actions systems is briefly given. Furthermore, Temporal Logic of Actions (TLA) is discussed. The next section describes Event-B formal language and techniques in detail.

2.7.1 Z

Z (Zed) (Spivey, 1989; Bowen, 2003) is a model-oriented specification language which is based on set theory and first-order predicate logic (Bowen, 2003). Standard set operators, set comprehension, cartesian products and power set are included in the set theory of Z. In addition, Bowen (2003) states that Z is just a formal specification notation that describes system functionality ("what") as opposed to the detailed implementation or execution of that functionality ("how"), while an abstract Z language is designed to express what is the required service or operation of the system. Thus, Z notation

is designed to create understandable expression for humans instead of the executable expression for computers.

All state spaces and their operations are described in Z notation schemas. A schema is divided into two parts, *schema signature* and *schema predicate*(Bowen, 2003). The schema signature as defined above the line is used to define the variables of the system which are system inputs, system outputs and internal state variables, while all possible system invariants and operations are specified in the schema predicate.

For the state space schema, whose structure is shown in Figure 2.1(a), the state of variable ($x_1 ... x_n$) can be defined with its types ($S_1 ... S_n$) in the top section. The bottom section is used to define the invariant for the condition or constraint of the state. Figure 2.1(b) shows a state space schema of *BirthdayBook*, a simple example adopted from (Spivey, 1989). The set of person's name and the set of corresponding birthday date are defined as basic types [PERSON, DATE].

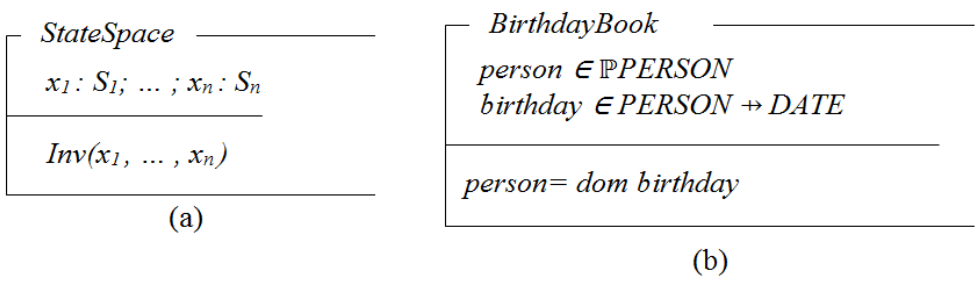


Figure 2.1: (a) The generic structure of a *StateSpace* schema (Bowen, 2003) and (b) state space definition of a lift control system

Figure 2.2 shows the structure of operation schema that gives examples of specifying the initialisation and state transitions. INIT illustrates the initialisation operation which indicates there is no birthday record in the *BirthdayBook* as shown in Figure 2.2(b). Figure 2.2(c) illustrates the operation to add a new birthday record. In Z specification, *StateSpace* in the operation schema expresses that the states defined in the *StateSpace* schema are changed after the operation. This results in the before (unprimed *StateSpace*) and after (primed *StateSpace*′) state predicates being defined. In this example, *BirthdayBook* defines four variables - *person*, *birthday*, *person*′ and *birthday*′ - in which the two former variables represent the state before the change, and the two latter variables represent the state after the change. To accomplish this operation, two inputs are required to illustrate the new birthday record that will be added into the birthday book. This includes the new person's name (*name?*) and the corresponding given birthday (*date?*). Note that in Z specification, the name of input is defined by ending with a question mark.

Z provides a refinement mechanism (Bowen, 2003). This is accomplished by removing the non-deterministic choice through refinement steps until the specification becomes closer to executable program code. Two kinds of refinement are provided by Z: *data*

Operation

$x_1 : S_1; \ldots ; x_n : S_n$
$x'_1 : S_1; \ldots ; x'_n : S_n$
$i_1? : T_1; \ldots ; i_m? : T_n$
$o_1! : U_1; \ldots ; o_m! : U_p$

$Pre(i_1?, \ldots , i_n?, x_1, \ldots , x_n)$
$Inv(x_1, \ldots , x_n)$
$Inv(x'_1, \ldots , x'_n)$
$Op(i_1?, \ldots , i_n?, x_1, \ldots , x_n, x'_1, \ldots , x'_n, o_1!, \ldots , o_p!)$

(a)

INIT

$person = \varnothing$

(b)

AddBirthday

$\Delta\, BirthdayBook$
$name? \in Person$
$date? \in Date$

$name? \notin person$
$birthday' = birthday \cup \{name? \mapsto date?\}$

(c)

Figure 2.2: (a) The generic structure of a *Operation* schema (Bowen, 2003) and (b) initialisation and (c) adding new birthday operation of a *BirtydayBook* example

refinement and *functional refinement*. Data refinement is the process that modifies the abstract data in the specification to the concrete data structure, e.g. array used in code, whereas functional refinement is the process in which the operations of the specification described in terms of predicates are refined to algorithms implemented in a programming language.

2.7.2 VDM (Vienna Development method)

VDM (Jones, 1990) was established in the 1970s by an IBM research group. Similar to other specification languages, VDM uses set notation and mathematical logic to build a formal model for a computer-based system. Furthermore, VDM also provides proof theories to guarantee that the proven specification is consistent. When designing the model with this specification language, proof obligations are created in terms of predicates. An example of such a proof obligation is *satisfiability obligation* (Müller, 2009) in which the result driven by an operation must satisfy the post-condition for every input that satisfies the pre-condition. These proof obligations are discharged to verify that the model is indeed the property of the system.

Figure 2.3 shows a VDM specification of *BirthdayBook* corresponding to the Z specification in Figure 2.2. Similar to Z specification, VDM models can be divided into two parts: state definintions and transitions. The state definition for *BirthdayBook* is illustrated in Figure 2.3(a). The invariant indicates that condition in which the set person and the domain of function birthday are the same is defined in an *inv* clause. State transitions are modelled as operations. Figure 2.3(b) shows operation *addBirthday* with the new person's name (*name*) and its given birthday (*date*) as the parameters.

This operation consists of three elements: external variables (**ext**), precondition (**pre**) and post condition (**post**). The operation can use the variable defined in the definition

part by indicating it in the **ext** clause. The **rd/wr** clause indicates the read/write external operation respectively. The preconditions express the condition before the operation occurs - for example, the new name must not already be in the birthday book, whereas the postcondition demonstrates the condition after the operation is applied - for example, the new birthday record that will be added into the birthday book. The (←) hooked variable denotes the state before the transition.

state *BirthdayBook* **of** *person* ∈ *PERSON*- **set** *birthday* ∈ *PERSON* $\overset{m}{\rightarrow}$ *DATE* **inv** *mk-BirthdayBook(p, bd)* ≜ *p* = *dom bd* **int** σ ≜ σ.*person* = ∅ **end**	*addBirthday(name:PERSON, date:DATE)* **ext wr** *person:PERSON*-**set** **wr** *birthday* : *PERSON* $\overset{m}{\rightarrow}$ *DATE* **pre** *name* ∉ *person* **post** *birthday* = $\overleftarrow{birthday}$ ∪ {*name* ↦ *date*}
(a)	**(b)**

Figure 2.3: (a) a data model and (b) operation *addBirthday* for a *Birthday* model in VDM

VDM starts to build a very abstract model of data type (Jones, 1990). By using VDM, system models are developed through Data Reification, i.e. refinement. The representation of data is chosen in which abstract data types are transformed into the concrete data types. This means that the data types of the concrete model are very close to the data type in the implementation languages. Operation decomposition is algorithm refinement in which the operation is decomposed into primitive constructs of implementation language (e.g. while loop). As a result the model containing the algorithm can be implemented in a computer language.

2.7.3 B-Method

In the mid 1980s, Jean-Raymond (Abrial, 1996) proposed the theory and methodology derived from Z for modelling computer systems, called **B-Method**. It has mainly been used in safety-critical systems such as railway control.

Modelling in B is based on the abstract machine and concepts of refinement. In an abstract machine, the variables are defined by using the set theory of sets, relations and functions. Predicate logic is used to specify the machine invariants. These invariants express the data-type information and encodes safety properties and requirements. Operations are specified to provide the state transition mechanism. Deterministic and nondeterministic state transitions are allowed in operations. Finally, proof activities which are *consistency checking* and *refinement checking* provided in B are used to check the B model. Consistency checking is used to check whether the operations of a machine satisfy the invariant. Refinement checking is the proof activity that is used to check that each refining machine is a valid and consistent model of the refined machine.

Figure 2.4 demonstrates some of the notation with an abstraction example. B-Method provides the explicit data declaration. Constants are defined separately from variables. Constants are specified in the CONSTANTS clause with the PROPERTIES clause describing properties that must hold for these constants. Furthermore, variables are defined in the VARIABLES clause with the INVARIANTS clause defining data type and safety properties used to guarantee whether these properties are always preserved by operations of the model. In Figure 2.4, for example, *person* and *birthday* are defined as variables in the VARIABLES clause with their properties expressing that the set person must be the same as the domain of function birthday. Operation *addBirthday* is also defined in order to add the new birthday record into the function birthday. After this operation, this defined invariant is checked by proof obligations to establish whether to check invariant holds. Figure 2.4 demonstrates some of the notation with an abstraction example. B-Method provides the explicit data declaration. Constants are defined separately from variables. Constants are specified in the CONSTANTS clause with the PROPERTIES clause describing properties that must hold for these constants. Furthermore, variables are defined in the VARIABLES clause with the INVARIANTS clause defining data type and safety properties used to guarantee whether these properties are always preserved by operations of the model. In Figure 2.4, for example, person and birthday are defined as variables in the VARIABLES clause with their properties expressing that the set person must be the same as the domain of function birthday. Operation *addBirthday* is also defined in order to add the new birthday record into the function birthday. After this operation, this defined invariant is checked by proof obligations to establish whether to check invariant holds.

(a)	(b)
MACHINE *m* ... **SETS** *n* **CONSTANTS** *k* **PROPERTIES** T **VARIABLES** *v* **INVARIANT** I **INITIALISATION** L **OPERATIONS** y ← *op*(*x*) ≙ **PRE** P **THEN** S **END;** ... **END**	**MACHINE** *BirthdayBook* **SETS** *PERSON DATE* **VARIABLES** *person, birthday* **INVARIANT** *person* ∈ ℙ (*PERSON*) ∧ *birthday* ∈ *person* ⇸ *DATE* **INITIALISATION** *person* := ∅ **OPERATIONS** **addBirthday**(*name, date*) ≙ **PRE** *name* ∉ *person* ∧ *date* ∈ *DATE* **THEN** *birthday* := *birthday* ∪ {*name* ↦ *date*} **END;** ... **END**

Figure 2.4: (a) Syntax of B abstract machine notation with (b) an example of a B Model for the *BirthdayBook* system

2.7.4 PVS (Prototype Verification System)

PVS (Owre et al., 1992) is a prototype verification system for creating formal specifications and constructing formal proofs. It was developed at the Computer Science Laboratory of SRI International in Menlo Park, California. PVS provides the powerful features that combines a specification language together with theorem-proving facilities.

The specification language of PVS is a higher-order logic language integrated with a rich type system. *Types* of PVS language is built on the usual base types e.g. *bool*, *nat*, *integer* and *real* and the function constructor [A ->B]. PVS also provides the complex type system including record, tuple and abstract data types e.g. stack. *Predicate subtypes* is an effective and powerful type system in which constraints are defined to the type; e.g. $nonzero : TYPE = \{n : real \mid n/ = 0\}$ denotes the subtype of nonzero real numbers. This benefits in type correctness verification. For example, the nonzero subtype can be applied to the division operation on real numbers which has the constraint of nonzero divisor. Type checkers enforce this division to ensure that the zero divisor is never applied. Proof Obligations (POs) called *type correctness conditions (TCCs)* are generated from type checkers and can be discharged by the *PVS theorem prover*.

A specification in PVS is structured by a collection of theories. These theories may be parameterised. A theory contains a theory name (*th_name*) with a list of formal parameters, an EXPORTING clause, an assuming part and a theory part as shown in Figure 2.5(a). The theory's name and its parameters must be unique identifiers. The EXPORTING clause creates availability to other theories. A name exported by a given theory can be imported into another theory by the use of the IMPORTING clause. The assumption part contains the top-level declaration that precedes the theory part. Normally, the constraint on the use of theory is defined as a formula in this assuming part. This can be viewed as the global invariant in which properties expressed in terms of assumption formulas are expected to hold in all instances of the theory. The theory part is the main body of a theory which normally defines key, necessary elements used in the specification including types, constants, variables, axioms and formulas. Invariants representing the correctness of the model behaviour can also be defined in this body part and it can be proved that they are always satisfied by the theorem prover.

An example of PVS specification is illustrated in Figure 2.5(b). This example indicates the formal specification for BirthdayBook which corresponds to the earlier mentioned specifications. Similar to earlier *BirthdayBook* examples implemented by other specification languages, two type names are defined in PVS: PERSON and DATE. Furthermore, type BIRTHDAY is also introduced as a function mapping from type PERSON to DATE. These types are used to define the variable person, date and birthday which represent the person's name, given birthday and birthday book. In order to add a new record into the birthday book, operation *addBirthday* is specified. Like programming languages, the operation in PVS uses **IF-THEN-ELSE** to check the condition before

the operation performs. In this case, operation *addBirthday* will check whether this new birthday record is already in the birthday book. The predicate *known?* is defined to check this constraint by taking a birthday book and a person's name and returning 'true' if that name already has birthday recorded in the birthday book. This condition leads operation *addBirthday* to not change the birthday book. However, if this new birthday record does not exist in the birthday book, the new birthday book will be returned in which the pair association of person's name (*name*) and birthday (*date*) will be added. Note that, in PVS, the WITH construct is similar to the function overriding in Z.

th_name **[<parameter>]:THEORY**
[EXPORTING]
BEGIN
<assuming part>
<theory part>
…
END ***th_name***
	BirthdayBook: **THEORY**
	BEGIN
	
	PERSON: TYPE
	DATE: TYPE
	BIRTHDAY: TYPE = [PERSON -> DATE]
	
	d0: DATE
	
	person: VAR PERSON
	date: VAR DATE
	birthday: VAR BIRTHDAY
	
	emptyBirthdayBook(person): DATE = d0
	
	known?(birthday, person): bool = birthday(person) /= d0
	
	addBirthday(birthday, person, date): BIRTHDAY =
	IF known?(birthday, person)
	THEN birthday
	ELSE birthday WITH [(person) :=date]
	ENDIF
	
	END *BirthdayBook*

(a)	**(b)**

Figure 2.5: (a) a necessary abstract machine notation with (b) an example of a B Model for the *BirthdayBook* system

2.7.5 Comparison of Z, VDM, B-Method and PVS

Z, VDM, B-Method and PVS are model-based formal specification languages that use mathematical notations to express users' requirements. However, Z is only a formal specification language that just provides a blue print at the specification phase of development (Lano and Haughton, 1995; Kaur et al., 2012). Unlike Z, the B-Method and VDM offer good choices for developers as development processes towards implementation. Furthermore, as claimed by Bicarregui and Ritchie (1995), the B-Method provides the syntax which increases a more programmatic feeling in algorithmic detail. In addition, the structure of the operation is similar to the structure of the function in the programming language. Furthermore, in B-Method, structuring of the specification is

one of the major features of this formal language. B-method creates the specification in a modular fashion as it provides modular mechanisms e.g. INCLUDES and IMPORTS, that enable subsidiary machines expressing different parts of the specification, to be combined to parent machines representing the full specification. With these benefits, therefore, B-Method can support the entire life cycle of software development from specification through to code generation (Lano and Haughton, 1995; Muñoz, 1999). Similar to B-Method, PVS provides a more programmatic structure e.g. IF-THEN-ELSE structure together with higher-order logic and proof checker provided by PVS, this specification language results in a rich specification model. However, PVS lacks the structural support to enable the process management of software construction in the way the B-Method does, as claimed in the study of Muñoz (1999).

The operation of Z, VDM and B-Method employ the same state transition mechanism. Z and VDM use the precondition and postcondition methods to demonstrate the state transition. In Z, the state after the change indicated by the prime notation ($'$) whereas the hooked notation ($\leftharpoonup$) is used to indicate the state before the change in VDM. On the other hand, B-Method provides the generalised substitution for state transition. This concept is similar to PVS which also use ":=" for substitution of state transition. This substitution has to be proved that the state before and after change must still maintain the invariant. The invariant maintenance in VDM is similar to that of B-Method and PVS in which it must be proved. However, unlike VDM, B-Method and PVS, Z defines an implicit invariant in the schema. Thus the maintenance of invariant is built into operation schema, and is proved by precondition calculation.

In conclusion, based on the comparison described above, Event-B language which is the descendant formal language of B-Method and inspired by Z and action systems is used in this work. The details of this formal language are described in Section 2.8.

2.7.6 Action Systems

Action Systems (Back and Kurki-Suonio, 1988; Back and von Wright, 1990) is a state-based approach that provides a method to program distributed systems. The behaviour of the systems is described as the possible interactions (actions) in terms of parallel and distributed execution, rather than a sequential execution. An action system ($\mathcal{A}$) can be expressed as a statement of form (Back, 1990):

$$\mathcal{A} \;=\; |[var\ x;\ S_0;\ do\ A_1 \,\Box \ldots \Box\ A_m\ od]|:\ z$$

where x denote the local variables and z denote the global variables of action system $\mathcal{A}$ in which these two variables are assumed to be distinct ($x \cap z = \varnothing$). S_0 is an initialisation statement indicating $x := x_0$ where x_0 is a list of intial values for variables

x. $A_1 \Box \ldots \Box A_m$ indicates the conjunction of actions ($A1 \wedge \ldots \wedge Am$). An action ($A$) is a description of a transition system. Each action e.g. A_i can be expressed in the form:

$$A_i = g_i \longrightarrow S_i$$

Where, g_i represents the *guard* and S_i is a body of the action. Guards are boolean conditions and constraints that enable the body to be executed. S_i indicates a state transition of variables, $x := x_1$ where x_1 is a list of updating values for variables x. Thus, when these guards are true, the sequence of statements on the state variables defined in the body will be executed.

Furthermore, action systems create the basic incremental manner step of refinement. The combination between refinement calculus (Back, 1980; Back and von Wright, 1990) and action system formalism enables the stepwise formal refinement for parallel algorithms. This starts with the specification of the system representing behaviour of the system as the form of a sequential statement. The parallel action system is constructed by making small refinements from the original statement, until this action system satisfies all constructed requirements.

2.7.7 Temporal Logic of Actions (TLA)

The temporal logic of actions (TLA) was introduced by Leslie Lamport (Lamport, 1994), which is mainly used to express and reason about the specification of the concurrent system. Basically, TLA is built on the combination concept between a logic of actions and temporal logic. This facilitates the system and their properties to be expressed by a logic of actions as well as the reasoning about their concurrent algorithms and behaviours to be performed by the temporal logic. TLA specification model is developed by TLA+, formal specification language aiming at the specification modelling and proof support. This specification language is based on set theory, first-order predicate logic and the TLAs.

In TLA, an action is a mathematical formula expressing variables, primed variables, and constants. The action also expresses the state transition representing the change of the current state to the next state. Like the state transition in Z notations, the current state is represented by the non-primed variables, whereas the primed variables represent the next state:- for example, the mathematical formula written as $x' = x + 1$ in which non-primed variable x denotes the current state of x before change and primed variable x' denotes the state of x after change (adding 1 to the current state).

A temporal formula of specification *Spec* expressed by TLA can be shown below (Lamport, 2000):

$$Spec \triangleq Init \wedge \Box[Next]_{<v1,\ldots,vm>} \wedge Live \tag{2.1}$$

As demonstrated in (2.1), *Next* indicates disjunction of all possible operations Op_i in specification *Spec* which is defined as follow.

$$Next \triangleq \bigvee_{i \in 1..n} Op_i$$

Each Op_i is basically a before-after predicate which is expressed over a set of variables $< v1, \ldots, vm >$. Op_i indicates the change of state between $< v1, \ldots, vm >$ and $< v1', \ldots, vm' >$. Thus, formula $\Box[Next]_{<v1,\ldots,vm>}$ means that every step (*Next*) in a trace is either a change of the state between the current states ($< v1, \ldots, vm >$) and the next states ($< v1', \ldots, vm' >$) by one of these operations Op_i or the unchanging current state after the state transition. The initialisation over these variables is defined by *Init*. Therefore, formula *Spec* is true, iff *Init* and *Next* are true.

To enable the specification assertion, the properties of the system are expressed in terms of invariants. The assertion "specification *Spec* guarantees invariant *Inv*" is expressed by the validity of formula $Spec \Rightarrow \Box[Inv]$. In TLA+, this can be defined as a theorem. This formula asserts that every behaviour of specification *Spec* must satisfy invariant *Inv*.

Furthermore, by using TLA, the liveness property, indicating "*something eventually does happen*", can be identified. As shown in (2.1), *Live* indicates a liveness property of specification *Spec*. This liveness property is normally specified with temporal operators $\Box(always)$ and $\Diamond(eventually)$. The general liveness condition for an action *Next* can be expressed as the *leads to* operator ($\rightsquigarrow$) (Lamport, 2002) as follow.

$$<Op_i>_v \rightsquigarrow <Op_i>_v \triangleq \Box(ENABLED <Op_i>_v \Rightarrow \Diamond <Op_i>_v)$$

This means that if Op_i ever become enabled, then a Op_i step will eventually happen.

2.8 Event-B

Event-B (Abrial, 2007) is a formal method for specification and verification of reactive systems. It is used to describe a system requirement specification which will be developed and verified by using set theory. Event-B has evolved from B-Method (Schneider, 2002) integrated with ideas of Action System (Back and von Wright, 1990; Back, 1990). On the one hand, the model built by Event-B is based on set theory and first-order predicate logic as B-Method is. On the other hand, stepwise refinement provided by Action System is enabled in the Event-B model (Kurki-Suonio and Järvinen, 1988). The simple abstract model is initially developed. Then, the additional properties are introduced in the later

derivation steps. This technique is well known as the stepwise refinement in Event-B. In addition, the correctness properties of models and their refinements are evaluated by proof obligations. This mathematical proof derived from a B model enables the correction by construction in which the consistency of each refinement level is checked.

2.8.1 Event-B Structure and Notation

An Event-B model consists of two parts: *machines* and *contexts* (Abrial, 2007; Hoang et al., 2011). The machine specifies dynamic elements or behaviours of the model. They are *variables*, *invariants*, *theorems*, *variants* and *events*. The static elements are *carrier set*, *constants*, *axioms* and *theorems* described by contexts.

The relationship between machines and contexts can be described as shown in Figure 2.6 (Abrial, 2010). Machines can see context(s). They can include one or several contexts. Moreover, a machine can be "*refined*" by another machine and a context can be "*extended*" by another one.

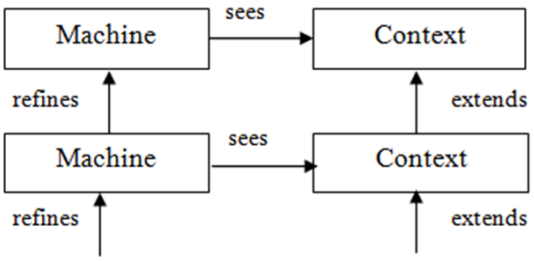


Figure 2.6: Machine and context relationships (Abrial, 2010)

Context Structure: Context structure (Abrial and Hallerstede, 2007) may contain carrier sets or constants representing the static parts of a model. Axioms may be defined in the context to describe the properties of those sets and constants. Contexts may include theorems that can be proved from the preceding axioms and theorems. Moreover, a context can extend several contexts and machines can see those contexts. The context structure is shown in the Figure 2.7.

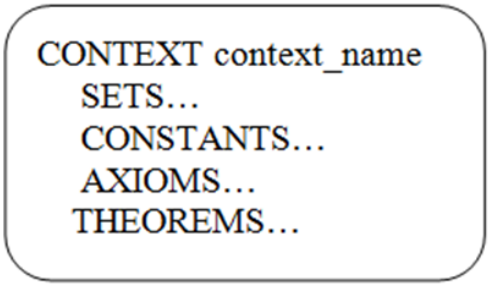


Figure 2.7: Machine and context relationships (Abrial, 2010)

Figure 2.8 illustrates an example of a simple context describing a static part of a birthday book. Similar to the former B-Method model illustrated in 2.4(b), PERSON and DATE are defined to be a set of person's name and date respectively.

```
context BirthdayBook_C0
sets PERSON  DATE
end
```

Figure 2.8: An example of a context structure defined as a static part of the *BirthdayBook* model

Machine Structure: The behavioral properties of Event-B Models are described by machines. Machine structure (Abrial and Hallerstede, 2007) consists of three distinct elements: *variables*, *invariants* and *events*. Variables define a dynamic element of state. A variable has a type which is a part of information expressed in terms of invariants. Furthermore, invariants also express the properties of a model.

Events are used to express the state transitions of the system. Like contexts, theorems must be derivable from invariants. Figure 2.10 shows an example of a machine expressing a dynamic part of the birthday book model. This machine includes the elements in the context *BirthdayBook_C0* by using "sees". In this example, invariant *@inv3* is the system property invariant stating that the set person must be the same as the domain of function birthday. In Event-B model, it must prove that every event must satisfy invariants.

```
MACHINE machine_name
  SEES context_name
  VARIABLES...
   INVARIANTS...
   THEOREMS...
    VARIANTS....
   EVENTS
  END
```

Figure 2.9: Machine Structure

The execution of events causes the change of states in the model. Each event contains a unique event name, guard(s) and action(s). A set of guards in an event expresses the necessary condition that enables the event to execute and a set of actions describes the change of states when the condition specified in the guards is hold (Abrial and Hallerstede, 2007; Hoang et al., 2011; Nakajima, 2009). An event can be illustrated in three different forms as shown in Figure 2.11. The form (1) is used to define the event which has vacuous true guards. Form (2) represents the event that has its guards and actions without parameters, while the last form (3) is used to define the event with its input local parameter t.

An example of different forms of events in Event-B model is illustrated in Figure 2.12. This figure shows an Event-B model for the lift system. On the one hand, Figure

```
machine BirthdayBook_M0 sees BirthdayBook_C0

variables person birthday

invariants
  @inv1 person ∈ ℙ(PERSON)
  @inv2 birthday ∈ PERSON ⇸ DATE
  @inv3 person = dom(birthday)

events
  event INITIALISATION
    then
      @inv1 person ≔ ∅
      @inv2 birthday ≔ ∅
  end

  event addBirthday
    any p d
    where
      @g1 p ∈ PERSON
      @g2 p ∉ dom(birthday)
      @g3 d ∈ DATE
    then
      @a1 person ≔ person ∪ {p}
      @a2 birthday ≔ birthday ∪ {p↦d}
  end
end
```

Figure 2.10: An example of a machine structure defined as a dynamic part of the *BirthdayBook* model

(1)	evt = **begin** S(v) **end**
(2)	evt = **when** G(v) **then** S(v) **end**
(3)	evt = **any** t **when** G(t, v) **then** S(t, v) **end**

Figure 2.11: Three possible forms of an event

2.12(a) shows the event with the parameter. The parameter f is taken as an input for the event *reqFloor* to determine any requests from any floors. On the other hand, the event without the parameter is illustrated in Figure 2.12(b), event *move* illustrating the movement of the lift when there are some requests at particular floors. In order to illustrate that the next floor is indicated randomly from the requested list, variable *nextFl* is assigned non-deterministically by event *move* from variable *reqFls* as shown in action *@a1* in Figure 2.12(b).

Considering the assignment action of events, there are three terms of generalized substitution which can be specified as an action of an event as displayed in Figure 2.13. The form (1) assigns deterministically the right-hand side value of the expression to the variable x, whereas the form (2) is a non-deterministic assignment which takes any values from the set involving parameter t and variable v. The form (3) assigns a value x', the

```
event reqFloor
    any f
    where
      @g1 f∈ fls
    then
      @a1 reqFls ≔ reqFls ∪ {f}
  End
```

(a)

```
event move
    where
      @g1 reqFls ≠ ∅
    then
      @a1 nextFl :∈ reqFls
end
```

(b)

Figure 2.12: An example of different forms of events in Event-B model (a) with parameter f and (b) without parameter

value after the occurrence of events, corresponding to t and v in the before-after predicate Q to the variable x (Abrial and Hallerstede, 2007; Hoang et al., 2011; Nakajima, 2009).

(1)	x := E(t, v)
(2)	x :∈ E(t, v)
(3)	x :| Q(t, v, x')

Figure 2.13: Three possible forms of generalized substitution

Proof Obligations (POs): POs are used to verify the properties of a machine by proving that every event maintains invariants. They are illustrated in terms of a sequence: "hypotheses" $\vdash$ "goal". This means that the goal must hold under the assumption of hypotheses. There are many proof obligations in Event-B. One of the most necessary proof obligations is invariant preservation (INV) as shown in Figure 2.14. The aim of this proof obligation is to ensure that each invariant in a machine is preserved whenever variable values are changed by the enabling event.

I(v) *(invariants)* *G(v)* *(guards of the event)* *S(v,v')* *(before-after predicate of the event)* $\vdash$ *I(v')* *(modified specific invariants)*	**INV**

Figure 2.14: *Invariant preservation proof obligation: INV*

2.8.2 Refinement in Event-B

Refinement in Event-B is a method that allows the model to be built in layers, separating requirements and design concerns. It starts with a simple abstract view of essential requirements. Each refinement step introduces more complex requirements and finally implementation, structure and algorithm. This refinement process completes when there are no remaining requirements that have not been considered from the requirement

document. The refinement in Event-B can be categorized into two types: *horizontal refinement* and *vertical refinement*.

Horizontal Refinement: this refinement provides the mechanism in which a model is built gradually by adding more features (Abrial, 2010; Yeganefard and Butler, 2010). This refinement approach does not consider implementation levels but concerns required behaviours specified in the requirement document instead. It starts with creating a very abstract model from the simple requirement. Then, in each later step, more and more requirements are introduced into the concrete model in order to make the model resemble as closely as possible the real system. By using this horizontal refinement approach, some refinement proofs are performed to make sure that the concrete model created by the later refinement steps is consistent with the model which has been made in the previous refinement steps.

Vertical Refinement: this refinement is also called "data refinement". Instead of considering adding more detail into the model, this vertical refinement approach is used to enhance the structure of the model so that it is closer to the implementation (Abrial, 2010; Yeganefard and Butler, 2010). By applying this vertical refinement approach, some states and transitions of the system are transformed into a form that can easily be implemented by computer languages. To accomplish this, the structure of a model is refined by adding algorithms through refinement steps. Furthermore, abstract data is transformed into concrete data structures. This means that some variables or abstract data structures can possibly be replaced by new concrete ones. As a result, gluing invariant, an important concept of vertical refinement, is introduced to link the concrete and abstract states.

POs for refinement: Two important refinement proof obligations are considered in the refinement approach, *Guard Strengthening (GRD)* and *Simulation (SIM)*. GRD PO aims to make sure that the guard of concrete events is stronger than the guard of abstract events, whereas SIM PO aims to ensure that each action in a concrete event can always correctly simulate the corresponding abstract event. Each action of a concrete event always correctly simulates some action of the corresponding abstract event. Note that although Event-B provides the distinct refinement approaches in order to improve process of modelling, POs of these two refinement approaches are not distinct.

2.8.3 Rodin as an Event-B tool

Rodin (Abrial et al., 2006; Butler, 2007) is an open tool platform based on Eclipse. This extendable tool was developed by the European Union ICT Project DEPLOY (2008-2012). The project's aim was to discover methods and tools which support the development of a complex and dependable software system. In addition, Abrial et al. (2006) claim that Rodin is designed as an open source so that multiple parties will achieve

the benefit in which they can extend and adapt the Rodin tool with their existing tools as a plug-in. They also believe that this would significantly increase the adoption of these tools by the software industries and the research in formal method. Good examples of plug-ins, which are widely used and installed in the Rodin tool, are ProB (Pro, 2008; Leuschel and Butler, 2008) and UML-B (UML, 2011b). On the one hand, ProB is an animator and model checker. It is used to validate the consistency and correctness of B machines. On the other hand, UML-B is the tool which provides the UML-like visualisation of Event-B modelling. It maps a graphical formal diagram to the Event-B language.

Rodin is used for formal modelling and proving in Event-B. It is intended to support seamless integration between construction, refinement and mathematical proof of the Event-B model. Rodin is also able to perform many tasks automatically and reply fast feedback to its users via well-designed and easy-to-use user interface (Abrial et al., 2006; Butler, 2007). For example, the label of variables, invariants and events are created automatically to enable easier modelling mechanism for software development. In addition, proof obligations corresponding to a model are generated and discharged automatically. For the proof obligation which is not discharged automatically, users can discharge manually by using interactive theorem provers.

Recently, ProB plugin 2.0[1](Bendisposto and Clark, 2014) for Rodin provides new Event-B Java-based API (Bendiposto et al., 2012; ADVANCE, 2013) that is used to access the states and events in Event-B machine. This new API also facilitates the Groovy scripting language to cooperate with Event-B API to create the programmatic abstraction containing the event trace of Event-B model (Bendiposto et al., 2013). This causes the sequence of events expressing the behaviour of the system to be executed automatically.

2.9 Decomposition and Refinement Atomicity in Event-B

2.9.1 Introduction to Decomposition in Event-B

As described in the earlier section, Event-B provides the incremental development mechanism in which software engineers can develop a model gradually by adding more precise details through stepwise refinements. This research considers two approaches to reduce the complexity of development and manage the structure of refinement: *refinement atomicity* and *decomposition*. Considering the former approach, the refinement atomicity approach enables an abstract event in an abstract machine to be refined into sub-atomic events in the concrete machine (Butler, 2008). Most sub-atomic events of a

[1]see ProB - http://www.stups.uni-duesseldorf.de/ProB/

decomposed abstract event are viewed as hidden events in the abstract machine. To develop these sub-atomic events, new events that are required to refine skip in the abstract machine are introduced in the refinement machine.

Decomposition, the later approach, splits a single machine into various sub-machines through the stepwise refinement. All these machines may either share some common state variables (Abrial and Hallerstede, 2007) or share events to synchronise each other (Butler, 2008, 2009). Since WSNs is a distributed network system where each sensor node communicates and coordinates with other nodes by passing messages via wireless environment, this research applies the *shared-event decomposition* to separate node controller model from the physical environment model. The following sections express the event refinement structure and shared-event decomposition approaches.

2.9.2 Refinement Atomicity in Event-B

To achieve the success of adopting refinement atomicity in Event-B, structuring the refinements associated with such atomicity decomposition is considered. Figure 2.15 demonstrates the simple atomicity approach originally proposed by Butler (2008, 2009) which is expressed by the diagrammatic notation of Jackson System Development (JSD) (Jackson, 1983; Jones, 1990).

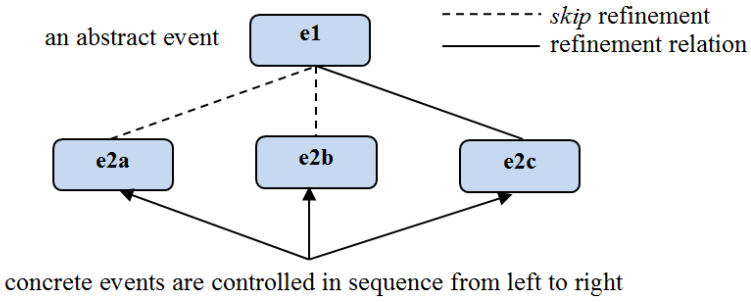


Figure 2.15: The event refinement diagram

Considering Figure 2.15, the root of the tree, *e1*, which represents an abstract event, is decomposed into three atomic sub-events in a refinement machine which are *e2a*, *e2b* and *e2c*. The sequence of these sub-events is read from left to right. For example, *e2a* and *e2b* are read in sequence or possibly done repeatedly before finally reaching the final event, *e2c*. This diagram also uses a dash line and a solid line to represent the type of relation between the root event and its children. The dash line is used to illustrate the skip refinement relation. For example, *e1* is linked to *e2a* and *e2b* by the dash line. This implies that *e2a* and *e2b* are the new events in the concrete machine which is invisible from the abstract machine. So, they refine skip in their ancestor machine. Another feature of the refinement diagram is the solid line. This line is used to describe

the refinement relationship. For instance, *e2c* is the child node of *e1* with the solid line. This indicates that *e1* is refined by *e2c* directly.

As mentioned above, the sequencing of sub-event execution is from left to right, so *e2a* and *e2b* should be performed in sequence or repeatedly before executing *e2c*. This can be accomplished by control variables introduced in the refinement machine. The events *e2a* and *e2b* are enabled by these variables in order to change the system state that causes the event *e2c* to execute.

Furthermore, this basic refinement atomicity approach is extended by Salehi Fathabadi (2012) to create refinement atomicity patterns. Two patterns have been applied in this thesis: the sequence and the xor-constructor. The sequence pattern has been illustrated as in Figure 2.15, while Figure 2.16 shows The xor-constructor diagram pattern.

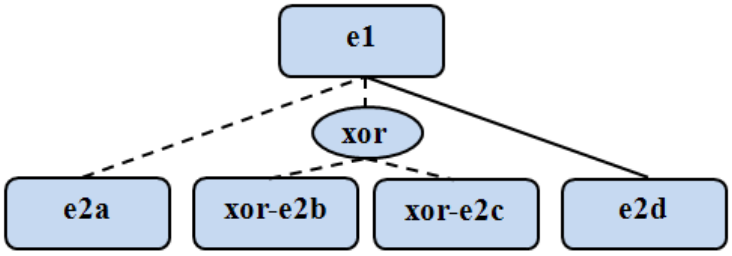


Figure 2.16: xor-constructor atomicity diagram patterns

Considering Figure 2.16, this starts from event *e2a* to be active. Then, the choice of events enables either xor-*e2b* or *xor-e2c* to be executed before event *e2d*, the refinement event of *e1* is executed.

2.9.3 Shared-Event Decomposition

The shared-event decomposition was proposed by Butler (2008, 2009). This decomposition style is based on the shared event. A single model is decomposed under the assumption that after the decomposition, the sub-models do not have the common shared variable and they communicate to each other by synchronizing between their shared events. Figure 2.17 shows the overview of shared-event decomposition, during the decomposition, the decomposed machine is created by splitting shared events such as *e2* into *e2a* in machine M1, Figure 2.17(a) and *e2b* in machine M2, Figure 2.17(b) and dividing the independent events such as *e1* from *e3* and *e4* independently.

The shared-event decomposition also supports the decomposition mechanism with message-passing. A single event is decomposed into two events which may have common parameters and common types which act as link parameters among the events. For example, the same parameter x can be passed by one event as an output parameter to another event as an input parameter. These events which share parameters can be split into several events.

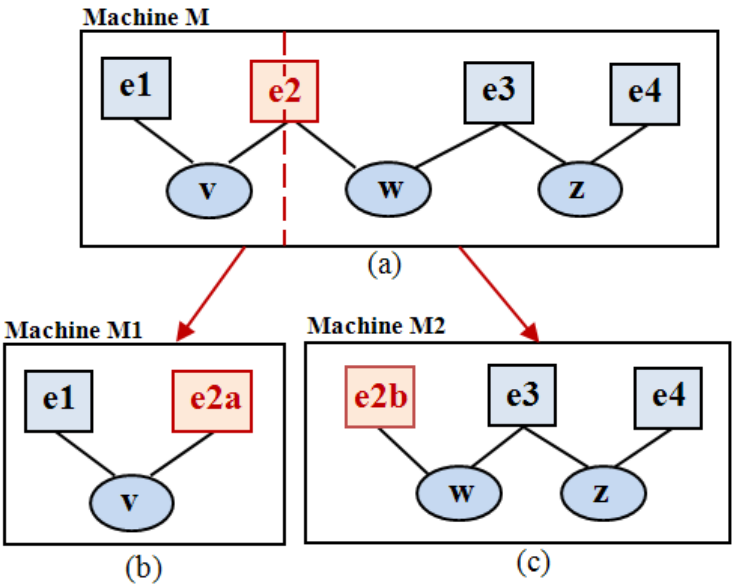


Figure 2.17: An example of shared-event decomposition

The Shared-Event Composition enables two sub-models that use the synchronising event to interact with each other to be combined. This results in the individual elements and their properties of these two composed models e.g. individual invariants, variables and synchronising events to be conjoined.

Figure 2.18 demonstrates an example of shared-event composition. In this example, two machines M1 and M2 representing sub-component specifications are combined as required. Events *e2* and *e3* containing independent variables (e.g. variables v and w used in these two events respectively) in machine M1 and M2 can synchronise in parallel. The result of the shared-event composition between these two machines is machine M where synchronising events *e2* and *e3* are composed: *e2* $\parallel$ *e3*. The result of composition enables composed events e2 and e3 to share their independent variables v and w.

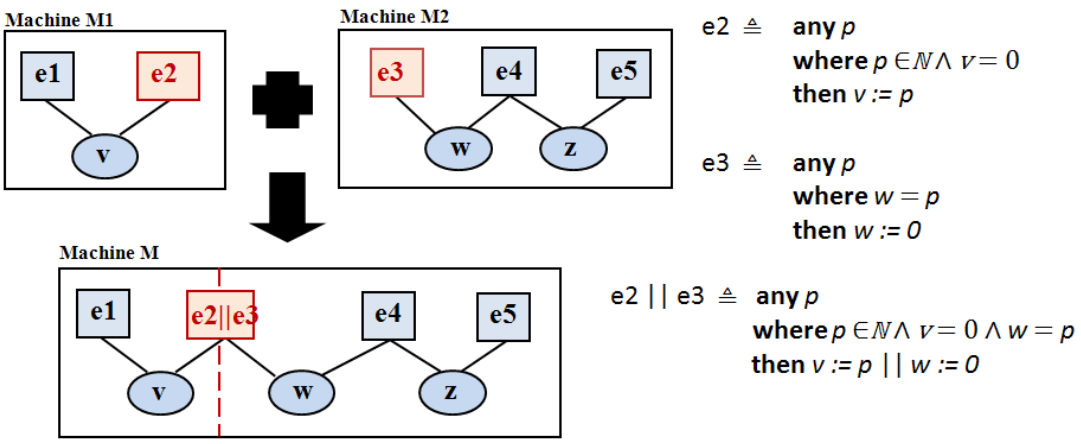


Figure 2.18: An example of shared-event composition

Shared-event Decomposition/Composition Plug-in: was developed by Silva et al. (2011). This composition plug-in supports the shared-event decomposition and composition as described above. Furthermore, this composition style also supports the combination of components with message passing through event parameters. This increases the benefit of reusability in Event-B modelling especially for distributed system.

2.9.4 Assessment of Decomposition in Event-B

The refinement atomicity in Event-B supports effective modelling of large and complex systems. This can solve difficulties caused by the complexity of proof obligations. In addition, by using the event refinement diagram to enable the decomposing mechanism in each refinement level, the sequencing of sub-event execution and the refinement relation are represented. A useful diagram hierarchy representing multiple refinement steps is provided and a control structure in the incremental hierarchy is created by this diagram. This leads to the improvement of the design of refinement in which software engineers can keep simply and easily adding more refinements. Consequently, the refinement diagram representing refinement atomicity enables more systematic and scalable refinement work (Butler, 2008, 2009).

Furthermore, the decomposition in Event-B reduced the complexity in the system modelling in which the problem or system is broken into the several parts that are easier to manage and maintain. This research focuses on shared-event decomposition technique which provides the modularity in the development of specifications. By using this decomposition technique, a model is partitioned into sub-component models that can possibly be further developed in parallel. These sub-models do not share individual variables but interact with each other by message-passing via shared event. In addition, this decomposition technique enables the proof obligations of the original model to be reused in the sub-component models. Shared-event composition technique provides the combination mechanism in which the sub-component models viewed as independent modules can be integrated through their interaction (via shared events) to create the full specification.

Therefore, as wireless sensor network systems grow and become larger and more complex, we plan to adopt refinement atomicity concepts to these various large and complex distributed systems modelling. Between each level of refinement, an abstract event is decomposed into a sub-event. Furthermore, as in WSN providing the distributed environment, each deployed sensor node interacts with the physical environments e.g. channel media, to exchange data between each other. Thus, to separate the software controller in sensor node from the physical environment, shared-event decomposition will be applied as it is suitable for developing message-passing distributed systems. We will describe more about the control variable in Chapter 6.

2.10 Model Checking

Model checking is a formal verification technique involving the exploration of all reachable model states and transitions to ensure that the system properties hold. It provides an automated verification in which the behavioral properties of finite-state systems are verified.

Model checking is based on temporal logic (Huth and Ryan, 2004; Rozier, 2011) which focuses on a notion of time. The temporal logic is used to create formulas describing behaviour properties or constraints of systems in terms of time. By using temporal logic, for example, we can express the statement of successful transmission as a transmitted message will eventually be received by the destination. Formally, the desired behavioural property including safety specification expressed by specification formulae in the temporal logic is checked to establish whether it satisfies the representation of the model by exploring all reachable model states and transitions. A counter-example is created when a sequence of states or traces leading to the violation of the specification formula is found.

One of the advantages of model checking is the automation of the analysis. The set of reachable model states are explored automatically by model checker in order to ensure the specification formula holds. A counterexample trace is automatically generated when the fault is discovered. However, as model checking supports the verification for finite-state systems, which have a number of finite states, the state explosion problem is the major reason for applying this technique to large and complex real-world systems (Clarke, 2008; Clarke and Klieber, 2012). For example, a concurrent system containing many processes and complicated data structures such as trees and recursive constructs can create enormous state spaces and finally result in an infinite number of states and transitions.

2.11 Model Animation

Animation is the validation technique in which the model can be traced step by step. This technique creates manual execution of user-defined short traces for sanity and requirement checking. This leads software engineers to discover whether the model they built has satisfied the requirements. Unexpected properties are also detected in the requirement document. Thus, animation facilities allow software engineers to achieve confidence that the requirement document is accurate (Leuschel and Butler, 2008).

2.12 Model-based Testing Approach for Formal Method

Trace checking became integrated into model-based or model-driven software engineering paradigms to strengthen the relationship between modelling and testing. In Howard et al. (2011), model-based trace testing is proposed to integrate tracing executable formal models and automated analysis into a model-based testing (MBT) framework. The traces are created and recorded from the execution of the system under test (SUT), developed by Java web-based development toolkits. These generated traces are run through ProB animator for B models and SPIN model-checker for Promela models to check them whether against the safety, liveness and deadlock of those models. Instead of generating test traces from SUT, our work generates abstract test traces directly from the Event-B controller interface which connects directly to the Event-B model. These are refined to concrete test traces for validating the model under test.

Malik et al. (2010) propose the application of a scenario-based testing (a kind of model-based testing) approach using Event-B stepwise development. Formal models, representing functional requirements and safety properties, are translated into Java to create SUT. Communicating Sequential process (CSP) is used to express test scenarios from Event-B models. These scenarios, then, generate executable JUnit test cases for testing generated SUT. This work is closest to our approach in which test scenarios are generated from Event-B models. However, event trace diagrams (ETDs) are used in our model-based trace testing (MBTT) approach rather than process algebra to express test scenarios from Event-B models as their graphical representation is useful for understandability. This is demonstrated in Chapter 8.

2.13 Conclusion

In this chapter, we have explained why formal methods are an important SE technology. This has also been explored with the analysis and discussion of the impact of adopting formal methods to industrial software engineering. Furthermore, formal verification and validation techniques such as proof-based formal methods, model checking and animation were also discussed and compared.

As formal methods use mathematical techniques to describe the requirements of the system and the model behaviour, one of the main advantages is the elimination of ambiguities and incompleteness of requirements.

Furthermore, the abstraction and refinement provided by formal methods can manage the complexity of the development by layering the abstraction of the models. Formal methods also provide decomposition in structuring developments in which the complexity of large system development is managed by breaking a single machine into sub-machines.

This leads to the consequential benefit that is an increase in the understanding of the system and the reduction of the chance of errors occurring in the later steps of development. Furthermore, demonstrating the absence of certain classes of faults that is impossible to reveal through testing is the key benefit of using formal methods.

Despite the benefit described above, there are a number of factors that have affected the application of formal methods to industry, limiting their success. One of the negative effects of failed adoption is the notation used in the formal language is hard to understand. The small-scale system development support from existing formal method tools is another failed adoption aspect. This challenge has influenced this research to contribute the co-simulation approach which integrates proof-based formal tools and technologies like Event-B language with its tool supports and development toolkits, Rodin, with simulation for WSN development as MiXiM. As Event-B language creates more programming structure which is very easy to read and its notations do not cost a considerable amount of time to get understanding for developers. Furthermore, Rodin, enables the graphic user interface and automatic environment, e.g automatic proof for development. In the later chapters, we demonstrate how Event-B can contribute these benefits to WSN engineering.

Furthermore, as we reviewed the various formal verification and validation techniques, we have discovered that state-space explosion of model-checking verification does not give the assurance of formal proofs. Consequently, this research focuses on a proof-based approach using Event-B to develop, verify and validate our case study model. We believe that the absence of certain classes of faults in network models will be guaranteed by POs. This is explained in Chapter 6.

Chapter 3

Wireless Sensor Network Background and Survey

3.1 Introduction

This chapter emphasises on the study of software engineering (SE) for wireless sensor network (WSN) development. The strengths and weaknesses of current development practices are also explored and discussed in order to support our proposed research contribution. Consequently, we start with illustrating the relevant background on WSNs. Section 3.2 explains the definition of WSN before describing basic sensor network architecture, hardware and software components in Section 3.3, 3.4 and 3.5 respectively. Various WSN application domains with some examples of successful applications are explained in Section 3.6. Then, in Section 3.7, we will outline the WSN development process. This section also assesses current practices on WSN development with the limitations we have observed. Section 3.8 expresses the comparison of simulations, emulations and lab testbeds before an overview of some existing simulators with their benefits against drawbacks is given in Section 3.9. To support our research contribution, Sections 3.10, 3.11 and 3.12 express literature reviews of model-driven engineering approaches, formal verification and co-simulation for WSNs respectively. Finally, in Section 3.13, WSN deployment problems, which we will tackle in development of our case study are discussed. The conclusion of this chapter is demonstrated in Section 3.14.

3.2 WSN Definition

A Wireless sensor network (WSN) is a wireless network consisting of computational sensor nodes (Choi et al., 2010; Stankovic, 2006). A sensor node is an embedded tiny device that works in a distributed environment. Each sensor node (Choi et al., 2010; Dong et al.,

2008) typically contains computing capability, sensing and wireless communication functions to perform monitoring of environmental conditions such as temperature, vibration, pressure or pollution over network topology.

3.3 Basic Sensor Network Architecture

A WSN consists of possibly a large number of sensor nodes which are densely deployed due to limited coverage and communication range affected by low power capabilities. Sensor nodes are deployed to a specific domain called a sensor field. Each of the distributed sensor nodes has the capability to sense and analyse data before routing such date either to other sensors or a (designated) data sink. A sink may be a fixed or mobile node which is able to connect to the sensor node in WSNs or to the internet where the reported data can be accessed by a user as shown in Figure 3.1.

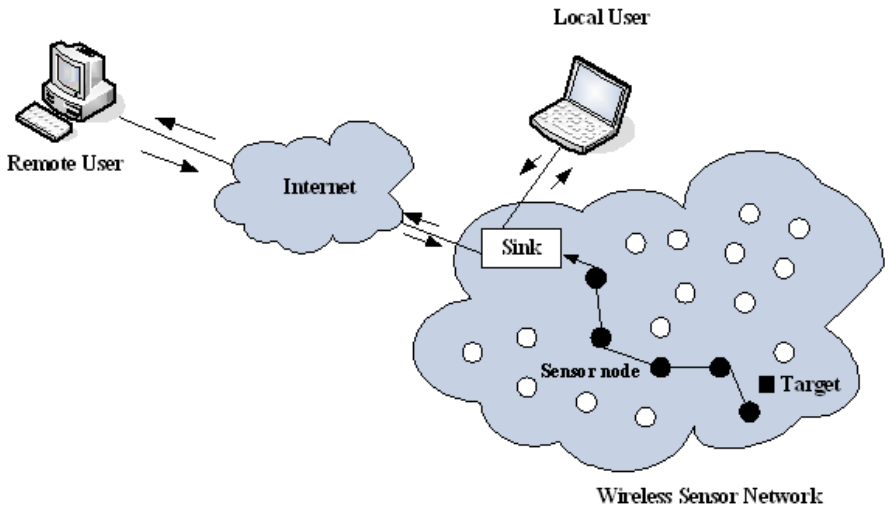


Figure 3.1: The multi-hop WSN architecture

3.4 Sensor Node Hardware Components

Each sensor node contains a set of embedded hardware with limited resources. A sensor node is composed of four basic components as shown in Figure 3.2 (Sohraby et al., 2007; Al-Karaki and Kamal, 2004): (1) a sensing unit composed of sensors that sense data from a particular phenomenon and ADCs that convert the analog signal to digital data; (2) a computing unit containing a small microprocessor for local data processing with limited memories e.g. RAM and flash memories; (3) a communication unit/transceiver used for interconnecting the sensor node to the network by sending and receiving radio signals; (4) a power unit supplying energy from limited energy resources e.g. batteries to all working component parts of the sensor node.

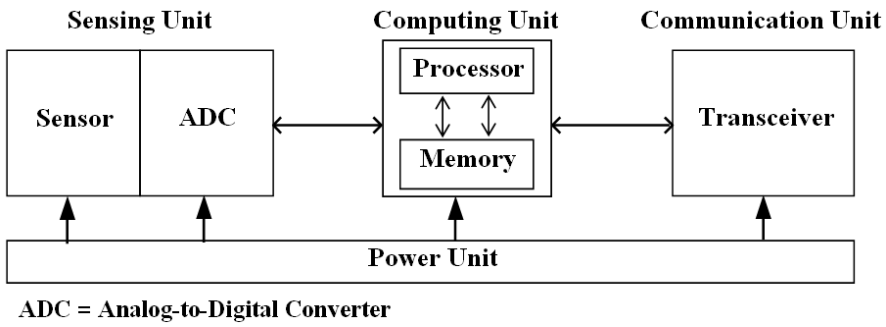


Figure 3.2: Overview of main sensor node hardware components (Sohraby et al., 2007; Al-Karaki and Kamal, 2004)

3.5 Sensor Node Software Components

Sensor node software components can be classified into two main classes: (1) operating systems including necessary drivers and middleware and (2) communication protocol stack containing protocol algorithms including application logics. The following sections describe briefly these software components.

3.5.1 WSN Operating Systems

For embedded systems like WSNs constrained by scarce resources e.g. limited memory and computational abilities, various specific and advanced operating systems (OSs) for WSNs, e.g. TinyOS (Hill et al., 2000) and Contiki (Dunkels et al., 2004), have been proposed. These specific purposed OSs are designed very lean in order to meet the limitation of resource usage and energy consumption (Reddy et al., 2009; Dunkels et al., 2004; Hill et al., 2000). In the OS context, the lightweight architecture is provided to support services and features under constraint awareness including drivers for the specific sensor node hardware platforms. As a result, OSs for WSNs are typically simple, providing the basic mechanism to access hardware, handle event interrupts and perform lower-power task scheduling

The WSN OSs usually co-operate with their associated programming languages which are typically C language and WSN-specific languages such as nesC for TinyOS (Hill et al., 2000) operating system. These OSs provide application programming interfaces (APIs), which allow the application to access the hardware through OSs. One example of these APIs is the nesC Active Message APIs[1] in the TinyOS. These interfaces provide basic communication primitives such as message sending and receiving mechanisms. However, most of these interface implementations are platform-specific which is normally needed to be bound directly to the radio hardware or to the specific MAC-level mechanisms

[1]see Active Message APIs - http://www.tinyos.net/tinyos-2.x/doc/nesdoc/telosb/ihtml/tos.interfaces.AMSend.html

deployed in such a hardware (Mottola and Picco, 2011). This leads programming for the real-world WSN deployments to be accomplished very close to the operating system and hardware platforms.

Furthermore, these underlying OS based languages affect the software code to be integrated with APIs provided by OS or hardware platforms into a single image. This results in the software code to be very difficult to understand and maintain. Therefore, in order to reduce the complexity of this monolithic development, various middleware approaches and technologies have been introduced such as Cougar (Yao and Gehrke, 2002), TinyDB (Madden et al., 2005), Mote-View (Turon, 2005), RUNES middlewares (Costa et al., 2007), TeenyLIME (Ceriotti et al., 2009) which provide component services interoperating among some software/hardware platforms. This enables the high-level programming abstraction in which reusable application components e.g. sampling and tasking and data collection components can be built abstractly from OS and hardware platforms. Thus, this benefits the reduction in programming effort.

However, few middleware abstraction technologies have been successfully widespread used in the practical development e.g. RUNE middlewares applied to implement the *Road Tunnel Application* (Costa et al., 2005, 2007). Most of them limit this success in adoption for two main reasons (Mottola and Picco, 2012). Firstly, since WSN applications have the wide range of characteristics and requirements, generic middlewares have been required in order to help developers match different requirements. This complexity causes some middlewares e.g. TinyDB to be very difficult to adapt to different needs as reported in Mottola and Picco (2012). Secondly, low-level system issues are still required to be focused for the development using some middleware technologies. For example, in the WSNs deployment for monitoring heritage building using TeenyLIME-based architecture (Ceriotti et al., 2009), the maximum memory usage constraint available on TMote Sky hardware is needed to be considered for the application development (Mottola and Picco, 2012). Another example is, from these middleware technologies: TinyDB (Madden et al., 2005), Cougar (Yao and Gehrke, 2002) and Mote-View (Turon, 2005) that they limit the development as they were assumed to be used for specific OS and hardware platforms as reported in Chatzigiannakis et al. (2007).

3.5.2 Protocol Stack for WSN

Figure 3.3 shows the network protocol stack used by a sink and all sensor nodes. This protocol stack consists of the following layers: *application layer*, *transport layer*, *network layer*, *data link layer* and *physical layer*.

Application layer: in WSN applications, this layer provides data sensing and processing, data aggregation and user query processing. Examples of well-known protocols in

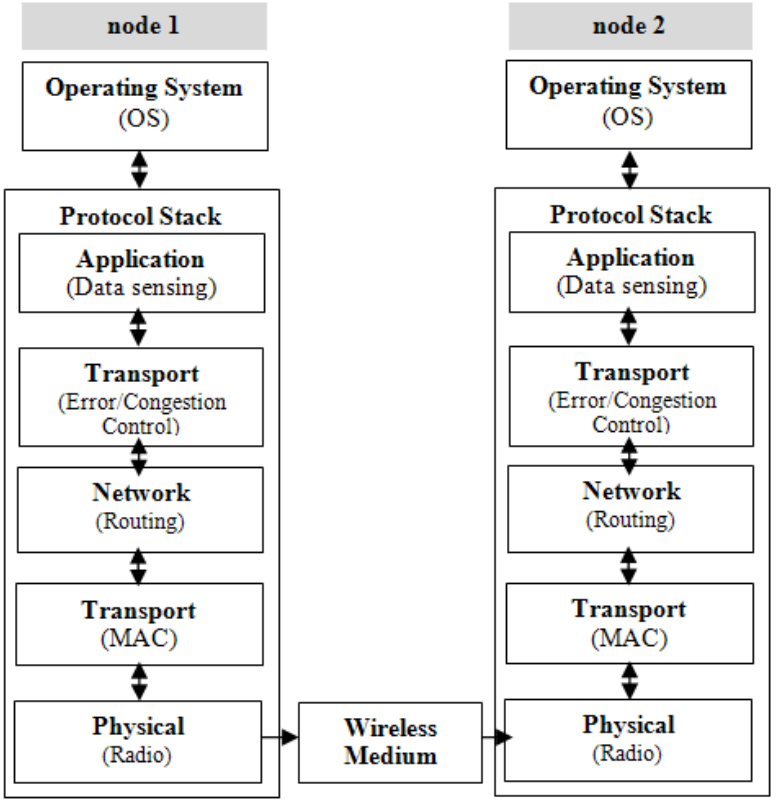


Figure 3.3: The sensor network protocol stack (Akyildiz et al., 2002)

this layer are sensor management protocol (SMP), task assignment and data advertisement protocol (TADAP) and sensor query and data dissemination protocol (SQDDP) (Raghavendra et al., 2004).

Transport layer: this layer bridges the application and network layers by maintaining the flow of data between these two layers. End-to-end communication services between the source and a sink are provided by this layer with reliable transport mechanisms including loss recovery, flow and congestion control mechanisms. This layer is also responsible to order message transmission by including a sequence number of the message header. The gap between the sequence numbers received at a sink can be used to indicate the message loss that is caused from either bit error or buffer overflow due to congestion. The transport control protocol (TCP) (Reed, 1980a) and user datagram protocol (UDP) (Reed, 1980b) are examples of transport protocols.

Network layer: this layer provides services for packet forwarding and routing through an intermediate sensor node. MintRoute (Woo et al., 2003) is one example of routing protocols implemented in this layer. This protocol creates the dynamic route tree of the sensor network indicated from every sensor towards a sink based on the link quality between nodes.

Data link layer: the data link layer is responsible for data stream multiplexing, data frame detection and medium access and error control. This layer also guarantees the

reliable point-to-point connection in the communication network. The acknowledgement mechanism can be used for this purpose. Furthermore, Medium Access Control (MAC) protocols are included in this layer where a shared access communication medium is managed. One of the main responsibilities of the MAC protocol is collision avoidance from interfering nodes causing energy waste. An example of MAC protocols is Sensor-MAC (S-MAC) (Ye et al., 2002) aiming to be an energy-efficient MAC protocol for WSNs. This protocol provides the periodic duty cycle time slots which enable each sensor node either to wake up or sleep at the same time. In addition, to address the collision avoidance issue, carrier sensing mechanism and Request to Send/Clear to Send (RTS/CTS) mechanism is provided by this protocol.

Physical layer: this layer contains networking hardware transmission technologies. It is mainly responsible for converting the bit stream from the upper layer into signals that are transmitted across the channel. This layer performs frequency allocation, carrier frequency generation, signal detection, modulation and data encryption.

3.6 WSN Applications

This section describes some WSN applications, in order to gain a better understanding of various WSN domains. It can be observed that WSNs have been adopted in a variety of areas from military to healthcare purposes (Choi et al., 2010; Sohraby et al., 2007). Traditionally, WSNs were applied to military application purposes (Chong and Kumar, 2003) such as sound surveillance systems; remote battlefield sensor systems; military command, communication and intelligence systems; weapon sensors; and target detecting and tracking systems. Recently, the interest in WSN applications has been expanded to support wide ranges of consumer applications and will tend to grow in the near future (Chong and Kumar, 2003; Akyildiz et al., 2002; Yick et al., 2008) such as infrastructure security and counterterrorism; environmental and habitat monitoring; weather sensing, building and structure monitoring; traffic control, the list becomes endless. The remainder of this section describes some successful WSN applications which are deployed in *Building Control and Monitoring*, and *Environment Monitoring domains*.

3.6.1 Building Control and Monitoring

Torre Aquila, a medieval tower in Trento (Italy) is one example of WSNs being applied in a Building Control and Monitoring domain (Ceriotti et al., 2009). Sensor nodes are implemented in the building to monitor the factors that may affect the building which can be temperature and humidity from environmental conditions and vibrations from a large group of people who visit this heritage building.

As shown in Figure 3.4, 16 sensor nodes and one sink (the node #0 placed at the top floor) were deployed in Torre Aquila. Three kinds of sensor nodes were deployed in the building as shown in Figure 3.4(a) (1) environmental nodes for environmental monitoring which are temperature, relative humidity and light. (2) deformation nodes for tower deformation study. (3) Acceleration nodes for vibration measurement. The sensed data from all sensor nodes is transmitted and recorded in a sink. This sink was also used as a gateway to interconnect with the front-end which allowed the remote user to be able to interact with the system.

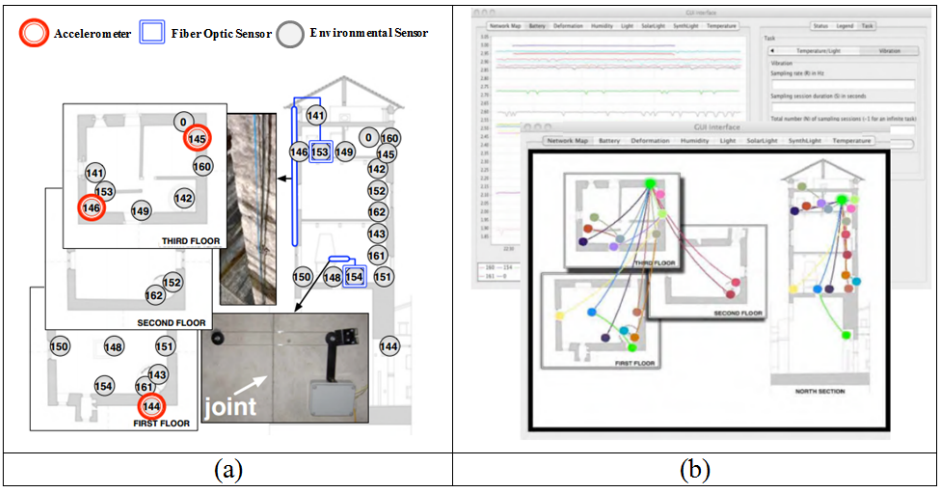


Figure 3.4: The Torre Aquila deployment, (a) deployment map and (b) graphical user interface (Ceriotti et al., 2009)

For analysis purpose, the structural engineer could analyse the gathered data collected by a sink through a graphic user interface (GUI), illustrated in Figure 3.4(b). This GUI displayed all gathered data recorded in a persistent database. Furthermore, it also showed the network topology where they were able to manage and control remotely the WSN.

3.6.2 Environment Monitoring

Environmental WSN systems are deployed in harsh and remote environments in order to monitor and analyse the environmental condition such as the volcano monitoring sensor system for assessing the hazards caused from the volcanic level by monitoring acoustic pressure and seismic velocity traces (Werner-Allen et al., 2006; Challen and Welsh, 2010), the Redwood system for monitoring the physical environment surrounding a coastal redwood tree (Tolle et al., 2005), the LOFAR-agro system to monitor climate factors e.g. humidity and temperature which might influence the disease in potatoes (Langendoen et al., 2006) and the SensorScope system for providing a standard of those mentioned environmental monitoring systems (Barrenetxea et al., 2008a,b).

SensorScope (Barrenetxea et al., 2008a,b) is an environmental monitoring system which is aimed at outdoor deployment. The environmental data is transmitted by the sensing stations to a sink through multiple hop. The base station (sink), then, uses the different standard gateway such as GPRS, Wi-Fi, or Ethernet, to forward all data to a database server. This project also provided Google Maps-based web interface and Microsofts SensorMap website for publishing data. In addition, the sink can possibly be managed remotely via GSM text messages as shown in Figure 3.5.

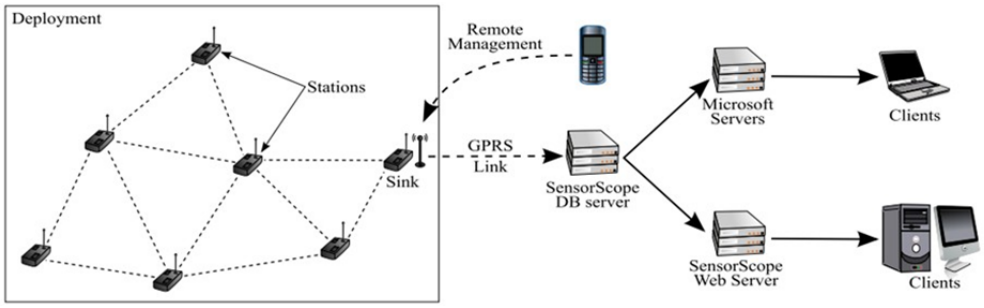


Figure 3.5: Example of sensor network deployment, SensorScope architecture (Barrenetxea et al., 2008a,b)

Six deployments were conducted with ranging size from 6 to 97 stations. One of challenging successful deployments was the Le Génépi rock glacier project in Switzerland. This project took place on the top of a rock glacier which caused dangerous mud streams during severe rains, resulting in flooding causing accidents on the adjacent road. In this project, 16 sensing stations were deployed on a 500x500 m area as shown in Figure 3.6(a). These stations were used to measure a micro-climate phenomenon in which the amount of rainfall was monitored and measured from soil water content and suction together with wind speed and air temperature. A sink was placed close to station 3 equipped with a GPRS module to transmit the packets to the database server.

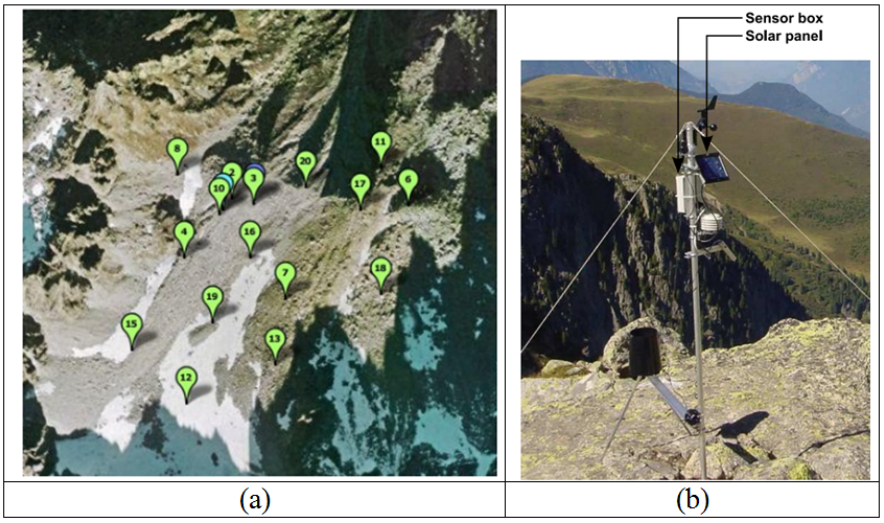


Figure 3.6: Génépi deployment, (a) deployment map (b) sensing station (Barrenetxea et al., 2008a,b)

3.7 Current Practices on WSN Development

3.7.1 Code-and-fix Development Process

In current WSN development, WSN software programmers apply a *"code-and-fix"* development process to create the WSN application (Picco, 2010; Mottola and Picco, 2011). This development process enables the source code to be produced immediately with minimal design. Programming needs to be very close to the specific OSs. After the development, testing is often processed at a late stage in the development cycle in order to ensure the developed code is free from bugs. The laboratory testbed may be conducted where the code is tested in the real node under pseudo real-world environment.

In this development process, the design of software has to be tightly coupled with the hardware platform which causes programmers to program their software code at the low-level. This results in programmers having to deal with low-level system issues as well as the design of distributed protocols e.g. routing and MAC protocols at the same time (Mottola and Picco, 2011). Therefore, this development process may influence the limitation which leads programmers' concentration to move far away from application logic including communication protocols and algorithms and finally causes the problem during deployment. For example, SensorScope project (Barrenetxea et al., 2008a,b), the real-world deployment project using this development style, led programmers to implement the embedded software encoding communication protocol stack together with the hardware platform. This affected the specific protocol algorithms, such as MintRoute and duty cycling MAC, to be designed very close to the low-level specific platforms like a Shockfish TinyNode sensor hardware and TinyOS operating system. Although, the laboratory testbed was used in this project, the malfunction in some sensor nodes was discovered after deploying in the real-world phenomena. This led programmers to extract the node from deployed environment to reprogram/correct the system as reported in Barrenetxea et al. (2008a,b). This may result in extremely expensive costs in the development.

3.7.2 Design for Deployment Process

Due to the wide range of characteristics and requirement of WSN application, WSN software becomes quite complex. As a result, Allen et al. (2010) proposed the design for deployment process for WSNs. It was derived after learning from deployment experiences from the different real-world deployment projects e.g. Volcano Monitoring Sensor Network (Werner-Allen et al., 2006; Challen and Welsh, 2010). The stages of the development process have been shown in Figure 3.7. This development process focuses on the application-centric view suggesting that the application requirement including the software and hardware functionality must be collected and analyzed carefully at early

stages. Then, design of the system including hardware, software and network is considered as a next step of WSN development. This stage also focuses significant attention to the selection of the algorithms and techniques for solving the problems specified in requirements. Furthermore, this design for deployment process provides distinct testing forms since more reliable powerful testing techniques are needed by WSN systems.

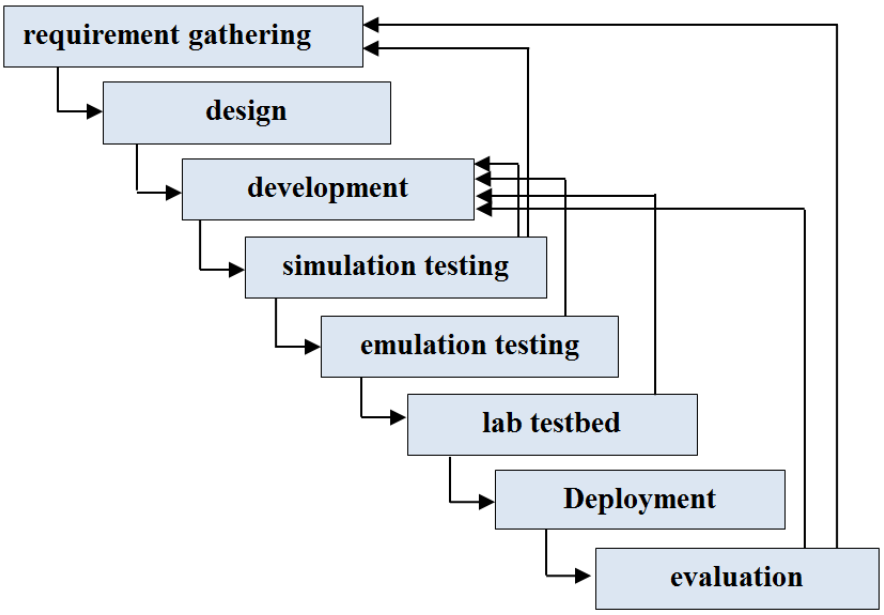


Figure 3.7: The design for deployment process (Allen et al., 2010)

Once implemented, WSN engineers perform testing phases. There are three testing techniques adopted into WSN development process: *simulation testing*, *emulation testing* and *lab testbeds*. Firstly, simulation aims to investigate and test network dynamic including communication protocols and algorithms for WSNs without considering the specific node behaviour. Therefore, simulation provides a higher level of abstraction of WSN systems which abstracts communication protocols and algorithms away from specific operating system and hardware platforms where other testing techniques e.g. emulation and lab testbeds do not. In current simulation practice, protocols and algorithms are layered to create a communication networking protocol suite by a standard protocol stack scheme. These are integrated with a stochastic environment framework of wireless channel, radio and analogue models to generate the long running testing scenarios. Thus, this testing stage provides a certain advantage, as wireless sensor engineers can discover any problems caused by the earlier stages. Furthermore, the result of network simulation is analysed and evaluated in order to conduct the performance of protocols and algorithms such as network latency and reveal the primitive network problem such as bottleneck.

Secondly, emulators aim to investigate the specific node behaviour. They are a hybrid approach combining software and hardware platforms. Typically, the independent software component achieved from the simulation stage is modified and integrated together

with the standard platform-specific libraries provided by operating system or hardware platforms. This results in the software code being targeted to a specific node. In order to investigate and test the node behaviour, the pseudo-nodes are created from the code together with generating the virtual dynamic network topology among these nodes and emulates them under controllable laboratory conditions. Therefore, this testing technique benefits the software component which can be tested based on real hardware assumptions. The node behaviour can be run and tested on programming modules on motes via a gluing API provided by WSN OSs. Various constraints on the node such as timing are also considered while emulating the code. If a problem is encountered during emulation, the wireless sensor engineers can backtrack to the design and implementation stage.

Finally, a laboratory testbed, a final testing stage, aims to validate the real node code with the real physical hardware under the real-world physical environment and connectivity. Therefore it provides the physical platform and environment which is similar to the actual deployment for WSN experimentation. The real node code achieved from the emulation testing phase is deployed and tested under realistic conditions. Real-world experiments are conducted in order to run the WSN network and provide feedback to wireless sensor engineers. As a consequence, this physical testbed reduces the gap between simulation and real deployment. As wireless sensor engineers can remotely configure, this is similar to actual deployment, they can backtrack to redesign and re-implement in case of failure.

Evaluation is a post deployment stage that analyses and evaluates the success of the deployment. After analysis and evaluation, some WSN engineers may redesign or re-implement their system because it performs against requirements, while others may review requirements again due to poor user requirements (Allen et al., 2010).

As explained above, the design for deployment process enables the benefit to the WSN development in which the developed application may meet the requirement including functional and non-functional requirements. However, the performance requirement e.g. sensing quality and network performance requirements is considered to be the first priority. To meet this type of requirement, the hardware specification together with the design of hardware configuration has to be focused. This may cause WSN engineers to not concentrate on the correctness of communication protocols and algorithms. Therefore, it is possible that the application still contains a fault that may influence the failure of meeting performance requirements. In the next section, we will discuss this issue in detail with the real-deployment project as an example.

3.7.3 Assessment of Current Practices on WSN Development

An important key issue for current WSN development is the high-level abstraction, the separation of concerns, powerful V&V techniques (Picco, 2010). This section describes the assessment of current practice on WSN development based on these key issues.

First of all, as can be noticed in the "code-and-fix" process which is used by the majority of deployments (Barrenetxea et al., 2008a,b; Mottola and Picco, 2011), recent WSN development process focuses on "low phases" of the development process. This can be confirmed by our literature survey of some existing deployment mentioned in Beutel et al. (2010). We found that few existing WSN developments and deployments deal with analysing and evaluating the correctness and performance of network protocol by applying simulations, emulations and lab testbeds. As claimed by Mottola and Picco (2011), there are few attempts focusing on high-level programming support for the real-world WSN development. Consequently, wireless sensor engineers lack SE methods, techniques and tools which support high-level abstraction at "early phases" of current practices on WSN development such as requirements and architecture specifications (Stankovic, 2004; Picco, 2010; Doddapaneni et al., 2012; Shimizu et al., 2012).

Although the design for deployment process discussed in Section 3.7.2 emphasises carefully on gathering application requirements, most practical WSN development focuses on non-functional requirements e.g. sensing quality requirement dependent on the hardware configuration, rather than the correctness of the protocol functionality and algorithms as we discovered in the real-deployment literature reviews (Allen et al., 2010; Barrenetxea et al., 2008a; Beutel et al., 2010). One of these examples is Volcano Monitoring Sensor Network (Werner-Allen et al., 2006; Challen and Welsh, 2010). This project evaluated the sensor network in terms of data fidelity (the quality and accuracy of the recorded data) and yield (the amount of captured data including acoustic pressure and seismic velocity traces reported by sensor network) from volcano phenomena. There were three iterative deployments in this project: *Volcàn Tungurahua (2004)*, *Volcàn Reventador(2005)* and *Volcàn Tungurahua (2007)*. In order to improve this performance evaluation and analysis mechanisms, the sensor board used in the previous deployment project (in 2005) was redesigned for the current deployment project (in 2007). Furthermore, WSN engineers of this project also adjusted and validated the software algorithm which contains the event-detection performing the algorithm. However, this correction and adjustment was accomplished based on different sample parameters configured on sensor hardware e.g. sample rates and per-sample resolution of seismic signal. They do not concern the correctness of the protocol algorithms in which we believe that this may affect the data yield e.g. loop problem in the routing algorithm (e.g. MintRoute protocol used in deployment project in 2005) as we reveal in this thesis. This leads to one of the challenges that is separation of concerns be needed in the WSN development.

As suggested by Picco (2010), in order to increase confidence in the correctness and performance of deployed WSN application, verification and validation must be considered at an earlier stage of the development process. SE methods and techniques emphasising automatic verification and validation at the early stage of the development process such as formal methods and model checking can increase confidence in the correctness of WSN application. The existing testing and debugging techniques for WSN development can detect the presence of faults but cannot show the absence of certain classes of faults such as safety property breaches. For example, in the routing algorithm, formal methods can guarantee by proving that there are no loops in the routing tree, while some testing techniques such as simulations cannot guarantee this property. Therefore, introducing formal methods at the early stage of the development can not only demonstrate the correctness of the WSN algorithms but also increase the level of program abstraction from formal specification, the most important part of formal development. The detail of these benefits will be described in Chapter 6.

3.8 Comparison of simulations, emulations and lab testbeds

As one of our contributions in this thesis is the separation of concerns and powerful V&V required by WSN development, this section compares the difference between simulations, emulations and lab testbeds based on a survey of Kiess and Mauve (2007); Imran et al. (2010). As can be seen in the design for deployment process mentioned in Section 3.7, we can summarise the relationship between each distinct testing form and other stages as shown in Figure 3.8.

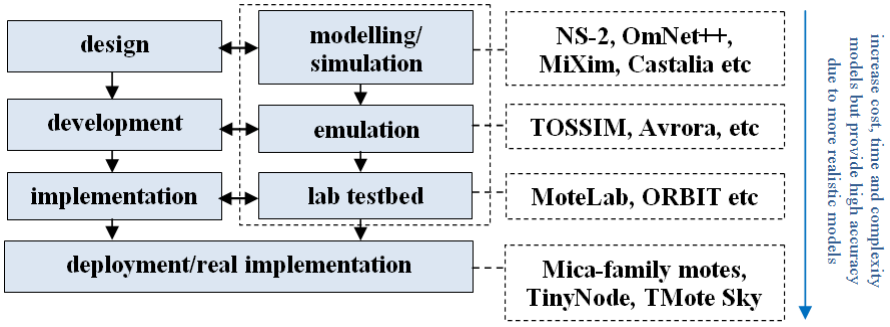


Figure 3.8: Summary of WSN development process

As mentioned in the previous section, simulation offers a good choice to be an early stage in designing, developing and evaluating of algorithms and protocols, since it provides an artificial software environment for simulating a model with a high degree of abstraction. For example, by using simulations, the decision of designing and developing routing protocols can be made independently from specific operating system and hardware platforms. Simulations provide certain benefits which are low cost testing, scalability and repeatability. The experiment can be investigated repeatedly with large scale networks

through different configurations. However, the model developed by simulation may not be accurate because of the lack of realism. The behaviour of the algorithm may not be reflected in a real implementation. Example of commonly used simulators are NS-2 (Fall, 2011), OmNet++ (Varga, 2011).

An emulator is a combination approach of software and hardware. The software component encoding the communication protocol and algorithm is integrated into the standard platform-specific libraries provided by WSN OSs or physical hardware platforms. Unlike simulations, therefore, emulations enable a certain degree of realism. The network algorithms can be run on an emulated OS environment under controlled laboratory conditions. The experiment can be performed repeatedly but with bounded scalability because the cost of development per node is more expensive than simulation as it needs the time to integrate the software with the API library provided by OSs or hardware platforms. TOSSIM (Levis et al., 2003) and Avrora (Titzer et al., 2005a) are examples of emulators.

In lab testbeds, the evaluation of algorithms and protocols is implemented under fully functional components with real-world settings. The WSN network is deployed and tested under realistic conditions. Real-world experiments can influence very useful feedbacks for simulation and emulations. However, the drawbacks of these real-world testbeds are the limitation of scalability and repeatability due to the high cost of hardware, software and human resources. Increasingly more realistic environments leads to an increase in the complexity of the model. As a result of this, it is very difficult to repeat experiments with real-world parameters. Examples of well-known lab testbeds are MoteLab (Werner-Allen et al., 2005) and ORBIT (Raychaudhuri et al., 2005).

For our research purpose, we are interested in the hybrid approach at the higher level of abstraction by combining simulation with formal models. As a result of this, a detailed discussion of simulation will be explained in Section 3.9.

3.9 WSN Simulators

As discussed in Section 3.8, simulation is an essential first step in the evaluation of algorithms and protocols for WSNs as it enables a high degree of abstraction. It enables the process of designing and constructing the protocol algorithm model and integrates such a model with the physical environment e.g. the radio channels. Furthermore, the experiments with the model are also constructed and reconfigured in order to evaluate algorithm performance, investigate and understand system behaviours and explain the interaction of WSNs and environments. Consequently, simulation is chosen to be one of the effective approaches and techniques to demonstrate this thesis contribution, aiming to enable the higher level of abstraction for WSN development. In Section 3.9.1, we discuss the classification of simulators before providing an overview of some WSN

simulators in Section 3.9.2. Finally, Section 3.9.2.5 discusses the comparison of the well-known WSN simulators, NS-2, OMNeT++, Castalia together with our target simulator used in this research, MiXiM.

3.9.1 Classification of WSN Simulators

WSN simulators can be categorised into **general** and **specific purpose simulators** (Imran et al., 2010).

- **General purpose WSN simulators:** this type of WSN simulator is a general purpose discrete event network simulator. It primarily aims to develop simulation models for standard wired and wireless communication networks. Examples of this classification are NS-2 (Fall, 2011; Altman and Jiménez, 2012) and OMNeT++ (Varga, 2011).
- **Specific WSN simulators:** for extra addition from the former categorised simulators, this type of WSN simulator aims to reduce the efforts in creating simulation models. It provides additional features for modelling specialized sensor networks such as sensing channel, sensor and battery models and specific protocol stacks for WSNs. Castalia (Boulis, 2011) and MiXiM (Köpke et al., 2008) are some examples of this category.

3.9.2 Overview of Some WSN Simulators

The WSN simulation tool which is used in this research is MiXiM. Hence, in this section, we explain some other simulation tools which provide similar functionality for creating WSN simulation models. As the experiment on the communication protocol algorithm implemented in the protocol stack is the main focus of this research, we only discuss simulators supporting the implementation of the protocol stack for WSNs.

3.9.2.1 NS-2

NS-2 [2] (Fall, 2011; Altman and Jiménez, 2012) is a discrete network simulator aimed at networking research. It provides features for creating a simulation model of TCP, routing and multicast protocols for wired and wireless networks. For mobile sensor networks, the simulator provides the support for 802.11 and 802.15.4 type of wireless MAC protocol. Especially, the later type is more suitable for sensor networks since it integrates basic modelling of energy consumption. Furthermore, sensor node routing modules supported by NS-2 also includes standard wireless IP ad-hoc network protocols.

[2] see NS-2 - The Network Simulator (ns-2) http://www.isi.edu/nsnam/ns/

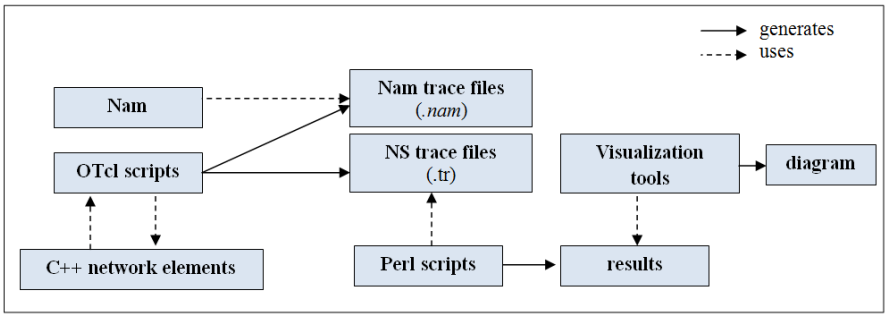


Figure 3.9: Development framework of NS-2 simulation

Currently, NS-2 has become very popular for sensor network development since it is used and maintained in academic research. As NS-2 is an open source network simulation and it provides an object-oriented design framework, it is easy to develop some extensions. As shown in Figure 3.9, NS-2 simulation models are written in the C++/C and OTcL languages. Protocol algorithms and simulation libraries are implemented by C++ files, whereas simulation environments and configuration interfaces are generated by the OTcL, an object oriented version of Tcl language. When running the simulation models, network simulation traces and packet traces are recorded into both the visualisation trace files (out.nam) and ascii trace files (out.tr). The former trace file can be viewed graphically by Network AniMator(Nam)[3] while running the simulation model. The Practical Extraction and Report Language (Perl) script is used to analyse and evaluate the simulation behaviour recorded in the latter trace file in which this result can be visualized by visualization tools provided by unix/linux such as *gnuplot*[4] and *xgraph*[5].

3.9.2.2 OMNeT++

OMNeT++[6] (Varga, 2011) is a component-based discrete event network simulation environment. The simulation primarily supports standard wired and wireless communication networks. However, some various extensions for WSN simulators exist such as MiXiM and Castalia. Like NS-2, OMNeT++ is an open source simulator based on Eclipse-based IDE and is popular in academic communities. Hence, OMNeT++ and its extensions are maintained by academic researchers.

OMNeT++ provides generic and flexible component architecture for models. Thus, it supports reusability of models by providing object-oriented programming language namely C++ to develop simulation models. Figure 3.10 shows the framework for OMNeT++ simulation development. Simple components (modules) illustrating network algorithms are programed in C++ (*.cc,.h)* before they are assembled into larger interface components and models using a high-level language like the Network Description

[3]see Nam - http://www.isi.edu/nsnam/nam/
[4]see gnuplot - http://www.gnuplot.info/
[5]see xgraph - http://www.xgraph.org/
[6]see OMNeT++ - http://www.omnetpp.org/

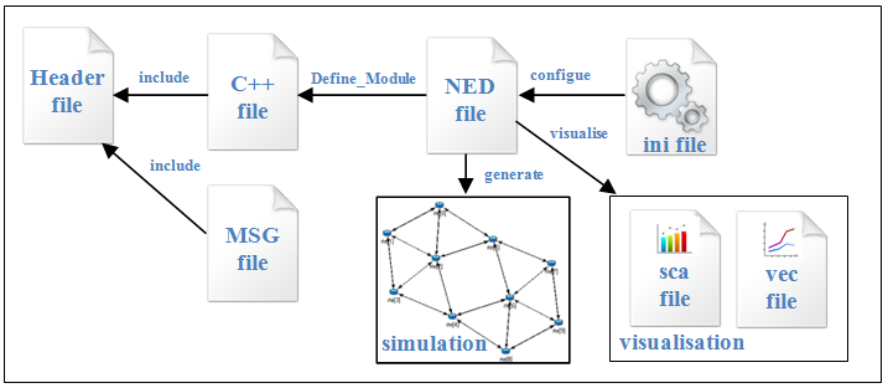


Figure 3.10: Development framework provided by OMNeT++ simulation

Language (NED). The NED language (*.ned*) is used to describe the structure of a simulation model including necessary parameters for simulation configuration (defined in *simple modules*) and network connection and gates (defined in *network modules*). NED files can be written using text editors and graphical interfaces. Message Objects defined in message files (*.msg*) are used to define message structures in networks. When the simulation starts running, the network topology defined in the NED file is read first. Then, it reads the configuration file (*omnetpp.ini*). This file contains the configuration setting that is used to control the execution of simulation. The output of the simulation is reported into result files: output scalar (*.sca*) and output vector (*.vec*) files. Furthermore, OMNeT++ has an Integrated Development Environment (IDE) for analysing and evaluating these result files in real time.

3.9.2.3 Castalia

Castalia[7] (Boulis, 2011) was developed by the National ICT Australia which aimed to provide the simulation framework for WSN, Body Area Networks (BAN) and generally networks of low-power embedded devices. It is based on the OMNeT++ platform. By using Castalia, researchers and developers can test their distributed network protocols or algorithms in realistic wireless channel and radio models as it provides advanced channel and radio models and extended sensing modelling provisions.

Figure 3.11 shows the basic module structure of Castalia. Each node connects to each other through wireless channel(s). The arrow indicates a message passing from one module to another module. The node sends its packet to the wireless channel which then decides the destination node for transmitting such packet. Each node is also linked through the physical modules. These physical modules indicate the physical environment where the nodes sample the physical reading.

[7]see Castalia -http://castalia.research.nicta.com.au/index.php/en/

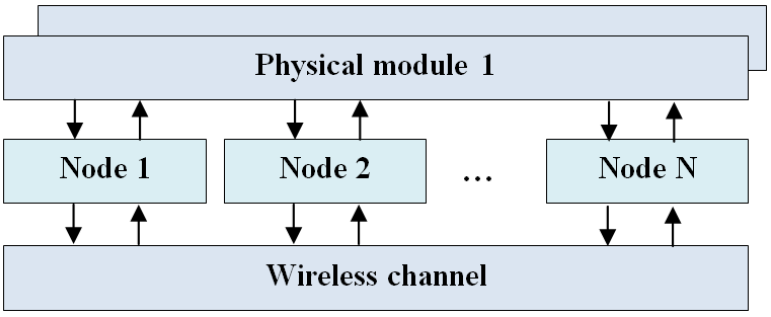


Figure 3.11: The modules and their connections in Castalia (Boulis, 2011)

Figure 3.12 shows the internal structure of the node module. As can be seen, most modules are connected to the resource manager by calling functions which are responsible to perform the operation for battery, CPU state and memory. This internal structure also enables the flexible framework for developers where they can develop their new algorithm. For instance, developers commonly change their application module by creating a new module to implement their new algorithm. Similar to this, the new modules for implementing new communication MAC and Routing protocols or mobility pattern are also usually changed by developers.

As Castalia models are developed based on OMNeT++, the OMNeT++ NED language as well as C++ language can be used to implement the structure shown in Figure 3.12. Normally, the complete hierarchy of these simulator files are dynamically loaded and processed by a feature of OMNeT++ without any recompilation of Castalia.

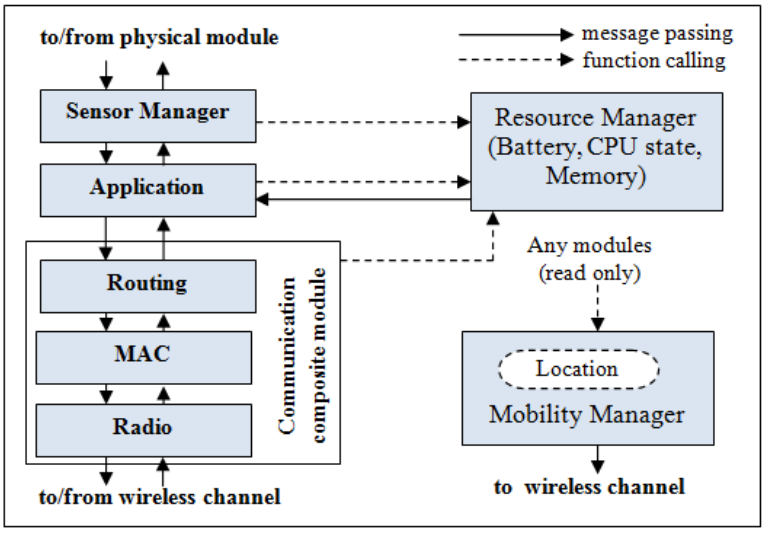


Figure 3.12: The node composite module (Boulis, 2011)

3.9.2.4 MiXiM

MiXiM[8] (Köpke et al., 2008) is an OMNeT++ simulation modelling framework. It was developed to create a more concise mobile and fixed wireless network model including WSNs and body area networks. It extends OMNeT++ simulation by adding several existing simulation frameworks to support WSN development including *Mobility Framework (MF)* for the mobility support, *connection management and general structure*, *CHannel SIMulator (ChSim)* for the radio propagation models and *MAC simulator*, *Positif* and *Mobility framework* for the standard protocol library.

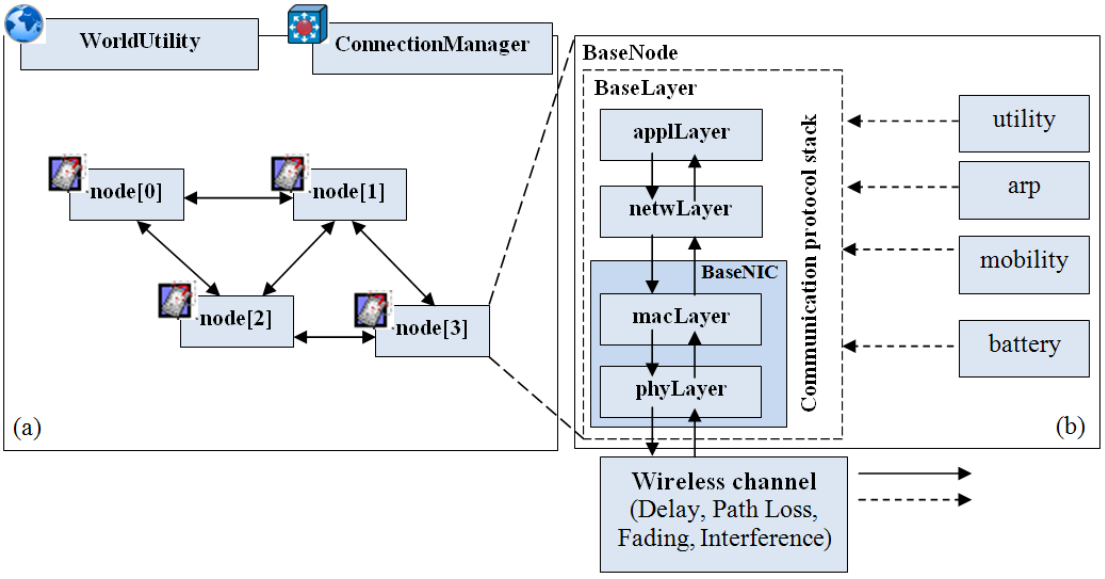


Figure 3.13: Node and Nic structure

Figure 3.13 shows the module structure provided by MiXiM which is divided into two perspectives: *simulation* and *node modules*. The structure of MiXiMs simulation networks has been shown in Figure 3.13(a). This perspective forms the network topology at the simulation time. The nodes in the network are generated instantly from the node module (*BaseNode*) consisting of the protocol stack as shown in Figure 3.13(b). Module *WorldUtility* creates the simulation playground which locates the generated nodes. The connection between each node is established by module *ConnectionManager*. Only nodes that are placed within the maximal interference distance of each other can be connected.

Like Castalia, the standard layers of the protocol stack model are implemented in the node module as can be seen in Figure 3.13(b). Those layers are namely the application (applLayer), network (netwLayer), MAC (macLayer) and physical layer (phyLayer). In order to illustrate tightly coupled design for the physical and MAC layer, the MAC layer is grouped together with the physical layer into the NIC (Network Interface Card) module (BaseNIC) as shown in Figure 3.13(b). Adjacent layers are linked together by

[8]see MiXiM -http://mixim.sourceforge.net/

OMNeT++ gates. These gates are used to pass up and down data and control messages between nodes such as the control message transmitted by the physical layer to indicate to the MAC layer that the transmission of a message is over.

3.9.2.5 Comparison of NS-2, OMNeT++, Castalia and MiXiM

The section compares the mentioned WSN simulators based on the study of Xian et al. (2008), Korkalainen et al. (2009) and Jevtić et al. (2009). Table 3.1 shows the comparison of NS-2, OMNeT++, Castalia and MiXiM. Three criteria, which influence the benefit for WSN development, are considered when comparing these WSN simulators: *development support*, *simulation environment* and *available models and protocols*.

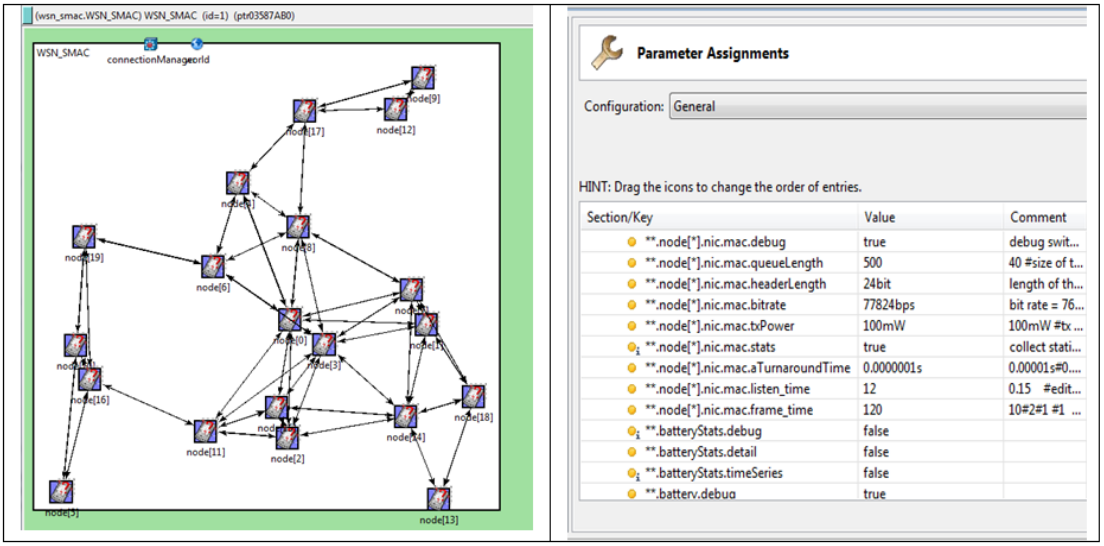
Development support: both NS-2 and OMNeT++ are open source simulators that have strong programmability. They are an object-oriented (OO) simulator. They provide a flexible framework with reusable components where new protocols or algorithms can be merged easily. However, OMNeT++ enables the advance benefit compared to NS-2 which provides NesCT, a programming language translator, to translate the code on the real node implemented by NesC programming language of TinyOS to C++ code for OMNeT++. This enables the communication protocol and algorithm based on the real code to be simulated. This benefit is also provided by the OMNeT++ extensions, Castalia and MiXiM.

However, the general purpose WSN simulators like NS-2 and OMNeT++ lack a concise modelling framework for wireless and mobility communication. As a consequence, there were some extensions to be developed for more realistic wireless and mobile simulation upon those simulators such as MiXiM and Castalia for OMNeT++. Castalia provides a similar development framework to that of MiXiM. Apart from this, MiXiM provides the graphical user interfaces of OMNeT++, instead of a command line user interface supported by Castalia. Thus, while running the simulation models, developers can view network simulation and packet traces graphically as shown in Figure 3.14(a). Furthermore, developers can also choose the right modules and configure values to their parameters through a graphical configuration interface provided by MiXiM as shown in Figure 3.14(b). Since MiXiM provides the flexible development framework, it has been chosen as the simulation for our research.

Simulation environment: NS-2 has limitations of scalability for large sensor networks. As reported by Xian et al. (2008); Korkalainen et al. (2009), the exponential simulation time is slowed down when the simulation is scaled up to a larger network. In NS-2 simulation experiments of Xian et al. (2008), for example, the execution time for processing 100 random REQ messages of the implemented directed diffusion protocol under the *IEEE 802.11 MAC* protocol increased considerably from 3000 to 9000 seconds when the number of nodes was increased from 1000 to 2000 nodes. For the random

Table 3.1: Simulation Comparison

	NS-2	OMNeT++	Castalia (based on OMNeT++)	MiXiM (based on OMNeT++)
Development support				
License	GNU GPL	academic, commercial	academic, commercial	academic, commercial
Sensor platform support	universal	universal, (TinyOS)	universal, (TinyOS)	universal, (TinyOS)
Simulation language	C++, C, OTcl	C++, C, NED, (NesC)	C++, C, NED, (NesC)	C++, C, NED, (NesC)
Simulation platform	FreeBSD, Linux, SunOS, Solaris, Windows (Cygwin)	Linux, Unix, MacOS, Windows	Linux, Unix, Windows (Cygwin)	Linux, Unix, MacOS, Windows
Simulation code exportable (protocols, etc.)	limited	limited , (NesC)	limited , (NesC)	limited , (NesC)
Simulation environment				
Scalability of large network (n>100)	fair (some cases)	good	good	good
Dynamic network topology change possible	yes	yes	yes	yes
Radom variable distribution	Pareto, Exponential, Normal, Long normal, Gamma	Pareto, Uniform, exponential, normal, Truncated normal, Gamma, Beta, Erlang, Chi-square.	Pareto, Uniform, exponential, normal, Truncated normal, Gamma, Beta, Erlang, Chi-square.	Pareto, Uniform, exponential, normal, Truncated normal, Gamma, Beta, Erlang, Chi-square.
Availability of a visualization	yes	yes	yes	yes
Available models and protocols				
Application support	none (sensing models and several application models via Mannasim)	several application models based on standard wired and wireless communication networks	Generic application model and Periodic Application Model	Generic application model and Periodic Application Model
Routing support	DSDV, DSR, TORA, AODV	fair (ad hoc routing)	fair (simple tree routing, multipath rings routing)	Adaptive Probabilistic Broadcast, Probabilistic Broadcast, WiseRoute, DummyRoute , Flooding
Standard MAC support	802.11, 802.15.4	802.11	802.15.4, 802.15.6, CSMA, T-MAC, S-MAC	802.11, AlohaMac, CSMA, L-MAC
Energy consumption support	yes	yes	yes	yes



(a) (b)

Figure 3.14: Graphical User Interface provided by MiXiM

distribution purpose, OMNeT++, Castalia and MiXiM supports more wide ranging random variable distribution compared to NS-2.

Available models and protocols: it is obviously clear that the specific WSN simulators provide more specific models and protocols for specialized sensor networks. For example, Castalia and MiXiM provide more specific MAC protocols like T-MAC, S-MAC and L-MAC for sensor networks, whereas only the generic standard MAC for wireless area networks such as IEEE 802.11 and IEEE 802.15.4 are provided by NS-2 and OMNeT++.

Furthermore, as can be seen in Table 3.1, NS-2 lacks an application model such as sensing models which is necessary for sensor networks to interact between application layers and network layers. *Mannasim*[9] framework based on NS-2 was developed for wireless sensor networks to provide a sensing model and several application models.

In conclusion, based on the comparison of NS-2, OMNeT++, Castalia and MiXiM as discussed above, we choose MiXiM to be our powerful target simulator for our research as it supports scalability for large sensor networks, more wide ranging random variable distribution and especially the implementation of the standard protocol stack compared to similar simulators. Furthermore, MiXiM also supports the flexible development framework in which the extendable base module (the general structure) provided by each layer of protocol stack enables the new communication protocols and algorithms to be developed easily. We will demonstrate this benefit in Chapter 5.

[9]see Mannasim - http://www.mannasim.dcc.ufmg.br

3.10 Literature Reviews of Model-driven Engineering Approaches for WSNs

In order to enable higher-level abstraction and separation of concerns for strengthening WSN development, model-driven engineering (MDE) techniques have been integrated into simulation (Doddapaneni et al., 2012; Shimizu et al., 2012; Wang and Baras, 2012). An architecture-driven analysis and development framework provided by MDE enables the beneficial multiple abstraction levels for WSN development which enables computation and communication algorithms, low-level hardware specification of nodes and the physical environment to be modelled separately. Typically, the abstraction of the model is expressed by using a Domain Specific Modelling Language (DSML) e.g. System Modelling Language (SysML) (OMG, 2012) in Doddapaneni et al. (2012), customising UML for a specific domain, used in Wang and Baras (2012). Then, this abstract model can be transformed into a more concrete (executable) model simulated in the simulation by the model transformation. The examples of target simulation are Castalia in Doddapaneni et al. (2012) and Mathlab/Simulink in Wang and Baras (2012). However, the abstract model produced by this integration approach is an informal model which is lacking in high-reliability V&V techniques needed by the WSN development in safety-critical domains. In addition, the generated code for simulation cannot guarantee that critical design errors will be discovered. Therefore, we can conclude that these approaches cannot guarantee the absence of certain classes of faults as formal methods do. The next section will discuss the study of applying formal methods to WSNs.

3.11 Literature Reviews of Formal Verification for WSNs

As mentioned in the earlier sections in which the distinct testing forms including simulation, emulation and lab testbed can improve the quality of the WSN application. These testing techniques can reveal the presence of errors in communication protocols and WSN software. However, these techniques cannot guarantee that the whole system is error free. Therefore, several formal verification approaches have been applied to WSN development in order to discover the hidden errors. This section reviews some existing verification techniques introduced in WSN developments.

3.11.1 Model Checking Verification

Since formal methods provide the benefit to wireless sensor engineers in which the ambiguities and inconsistency of the system are eliminated, a number of model checking approaches are applied to WSN (Nair and Cardell-Oliver, 2004; Fehnker and Gao, 2006; Ölveczky and Thorvaldsen, 2007; Schuts et al., 2009; Sharma et al., 2009; Saghar et al.,

2010) The following describe examples of various model checking approaches and techniques applied to WSN development.

A successful model checker employed in this field is UPPAAL (Behrmann et al., 2004) e.g. for the clock synchronization algorithms in gMAC (Schuts et al., 2009) and Denial of Service (DOS) protection in arrive routing protocol (Saghar et al., 2010). Typically, the invariant properties including safety, liveness and deadlock freedom properties are defined in terms of LTL (Linear Temporal Logic) properties in order to check and confirm that these properties hold in the formal model. One example of these properties is the safety property defined for the clock synchronization algorithms in gMAC (Schuts et al., 2009) ensuring that *"when a node is sending a packet, all its neighbours are always listening in order to receive such a transmitted packet"*. Similarly, SPIN (Holzmann, 2003) is another most widely adopted model checker for WSNs.

Furthermore, since the unreliable or unpredictable behaviour exhibited by physical environment, probabilistic model checking has been applied throughout the performance analysis of various network protocols. This class of model checking not only checks the correctness of protocols and algorithms but also analyses the performance of such a protocol and algorithm by simulating with probabilistic behaviour. For instance, in the work proposed by Ölveczky and Thorvaldsen (2007), RealTime Maude (Ölveczky and Thorvaldsen, 2007) is used to model, simulate and further analyse the formal specification of OGDC algorithm. In this work, pseudo-jobs were generated by this specification model in order to simulate the probabilistic behaviour by Monte Carlo Simulation. Time-bounded reachability analysis and temporal logic model checking were also used to verify all possible behaviour generated by the pseudo-random generator. Other similarities to this work are Real-Time Maude (Ölveczky and Thorvaldsen, 2007) for LMST (the local minimum spanning tree) topology control algorithm (Katelman et al., 2008); PRISM for Flooding and gossiping protocols (Fehnker and Gao, 2006) and Fault-Tolerant Target Tracking (FTTT) (Bhatti et al., 2010).

Based on this literature review, we have discovered that most model checking research for WSN development focuses on model verification together with performance analysis as some model checking languages can simulate the formal model. This benefits the WSN development in which the developed system may satisfy users expectation including both functional and non-functional requirements. However, the fundamental problem of model checking is state-space explosion (Clarke and Wing, 1996; Valmari, 1998). From our perspective, as WSN systems consist of a considerable number of nodes, this may impact the systems to contain a large number of states which tends to increase exponentially due to the growth of the number of processes and variables. This causes the process of model checking to be time consuming. Although various methods and techniques have been introduced to reduce the state explosion problem in model checking e.g. the heuritics which select and explore the interesting portions of the state space (Clarke and Wing, 1996; Valmari, 1998; Mus, 2004; Wang et al., 2009), this creates

the limitation of model checking which restricts the network instances and temporal properties to be small.

3.11.2 Proof-Based Formal Verification

Similar to model checking verification, there are several work studies that have been dealing with proof-based formal methods for network protocol verification in the past few decades. Examples of these are HOL for distance routing protocols (Bhargavan et al., 2002); Event-B for DSR protocol (Méry and Singh, 2011a) and ZigBee Protocol Stack(Gawanmeh, 2011); Prototype Verification System (PVS) for WSN communication protocols (Bernardeschi and Masci, 2008) and NDlog program implementing the path-vector protocol (Wang et al., 2009); and formal analysis for critical element detection (Matousek et al., 2010).

Traditional proof-based formal development and verification mainly focuses on the development of mathematical models for distributed protocols and algorithms as well as the proof of the model correctness. In Bhargavan et al. (2002), for example, an interactive theorem prover HOL has been used to verify distance routing protocols e.g. the Ad-Hoc On-Demand Distance Vector (AODV) protocol (Perkins and Royer, 1999). The various theorem properties have been defined for the proof of correctness of protocols e.g. the property guaranteeing the loop-free in the route tree for AODV. Similarly, Event-B has been applied to enable the incremental development and verification of IEEE 1394 tree identify protocol (Abrial et al., 2003), DSR protocol (Méry and Singh, 2011a) and ZigBee protocol stack (Gawanmeh, 2011). Most invariant properties created in these research studies were properties guaranteeing reliable network connection and communication, as well as the correctness of protocol algorithms. For example, in Méry and Singh (2011b), the safety property for network communication was also specified e.g. the property guaranteeing that *"the transmitted data packet will be either received successfully by the destination node or lost due to network failure"*. Gawanmeh (2011) defined the network property for ZigBee protocol stack as an invariant stating that any node cannot join the failed network.

Despite the powerful V&V enabled by the traditional proof-based formal methods in the WSN development as expressed above, the executable formal model is required in order to analyse and evaluate the network performance. As a result of this, there are some research studies applying the executable formal methods to WSNs. (Bernardeschi and Masci, 2008; Wang et al., 2009; Matousek et al., 2010).

In Bernardeschi and Masci (2008), Prototype Verification System (PVS) was adopted to specify and simulate WSN communication protocols. The correctness properties of routing algorithms were verified by typechecker and theorem prover before generating

executable LISP code to simulate and analyse performance properties such as robustness of topology changes and energy consumption. Furthermore, by simulating this executable formal model, the reliable network property regarding loss of packet due to buffer overflow is analysed and evaluated. However, in this paper, the authors did not explain fully about the proof construction process to illustrate how all desired properties always hold. They only verified the type checking e.g. *"the sensor node id indicated by a unique natural number that must be greater than a given total number of nodes"*.

Another comparable work is DNV (a declarative network verifier) (Wang et al., 2009) which provides the integration approach of network specification, protocol verification and implementation within a declarative framework. This work benefits the interworking approach between the formal development in PVS and the declarative network specification in NDlog, a logic-based recursive query language for querying network graphs. In this work, the protocol algorithm e.g. the path-vector protocol was encoded in NDlog program. The algorithm property and invariant were expressed in terms of NDlog rules in order to enable the NDlog program to be tested and validated along the execution. Then, the specification expressed in NDlog code together with NDlog rules was generated automatically into the formal specification and theorems in PVS respectively. Consequently, this resulted in the formal verification process in which the invariant e.g. the route optimality property guaranteeing the best path for routing are formally proved by PVS. However, unlike our research work, this work influences the reverse engineering process in which the program specification of sensor network encoded in a NDLog program is translated into a PVS model. This work lacks the high level of abstraction in order to reduce the complexity of the development. In our research, we introduce forward engineering that starts with implementing the high-level specification by using Event-B, the target formal language. The refinement technique provided by this formal language also influences the benefit in which the complexity of development is reduced.

The work proposed by Matousek et al. (2010) is closely related to our work. They proposed a combination approach for detecting critical network elements e.g. the link and node failures by using formal analysis based on set theories and indicating time dependencies by using the OMNeT++ simulator. The reachability property was defined to analyse the survivability of the network connection. One of the necessary properties is the property indicating unreachable destination. This can be detected by finding that if there is a broken node or link in the available path for data transmission between the source to the destination, then the destination is unreachable. Similar to other work, the authors did not demonstrate any proof to illustrate how all desired properties always hold. Their research focused on general network domains as opposed to WSN domains.

Overall, we can conclude that similar to model checking for WSNs, the proof-based formal method is required to create the executable formal model for WSNs in order to analyse and evaluate the performance. Typically, this can be accomplished by generating the formal specification model into a simulation code. However, we have discovered that

most research studies focus on using these executable formal methods to measure the reliable network performance. There are few attempts to use these proof-based formal methods to prove functionality, safety and liveness properties. Therefore, our challenge is to strengthen the V&V of software for sensor networks by integrating proof-based formal methods together with WSN simulation. This integration framework focuses on not only the network performance analysis and evaluation as other executable formal methods do but also how proof-based formal methods can be used to construct the proof of correctness of protocols and algorithms by using Event-B language before deploying the executable formal model in the co-simulation framework.

3.12 Literature Reviews of Co-Simulation for WSNs

A hybrid design framework is needed for WSN development when the WSN application containing communication protocols and computation algorithms closely interacting with physical environments has become more complex. Therefore, several hybrid system modelling and simulation approaches and techniques enabling co-simulation for WSNs have been proposed. This section reviews and discusses the literature and study of co-simulation approach for WSNs which have been divided into two categories: *model-based* and *fomal-based co-simulation*.

3.12.1 Model-based Co-simulation

Since model-based software engineering provides the advantage in which the program code executing on a hardware/software platform is generated automatically from the model, several research studies have been proposed to the WSN community. The model-based system design (MBSD) framework for WSNs proposed by Wang and Baras (2012) is one of these examples which co-models and co-simulates event-triggered components representing network algorithms together with continuous dynamic behavior exhibited by physical environment. This framework provides co-design perspectives. One is the system behaviours indicated by event-triggered components to be modelled by using System Modelling language (SysML, based on UML). Another is physical environments exploited by continuous components to be modelled by Simulink or Modelica and their behaviours to be expressed by differential equations. Furthermore, Simulink and C++ source code can be generated automatically from these two perspective models in order to construct the co-simulation and evaluate the performance. The extension to this work, HybridSim (Wang and Baras, 2013), adopts FMI standard to co-simulate between sensor application models of TinyOS expressed by SysML and simulation environment generated by Modelica. Their work is similar to ours in which the model of the node is co-simulated with the environment. However, their node models do not contain formal elements that leads to a lack of formal precise semantics.

Another model-based co-simulation framework that is the closest to our co-simulation framework is NMlab (Heimlich et al., 2010) which provides a co-simulation framework for Matlab and ns-2 simulator. The system controller implemented in Matlab co-simulates with the network models provided by ns-2 by using socket interfaces. Similar to this, HarvWSNet (Didioui et al., 2013), a framework for energy harvesting WSNs, combines the strengths of two development toolkits via a standard socket interface. The power management model is implemented in Matlab to communicate to the wireless sensor network communication model provided by WSNet. However, the communication algorithms at each protocol layer are still implemented in WSNet simulator.

3.12.2 Formal-based Co-simulation

Formal-based Co-simulation approaches have been introduced to WSN development for two reasons. One reason is to enable the executable formal model in which not only the communication protocol and algorithm is verified by the highly reliable formal verification but also the performance of protocol for wireless network is analysed and evaluated through the long running simulation as already expressed in the earlier section. The other reason is more concrete physical environments containing the realistic physical environment e.g. wireless channels, the radio and analogue models is needed to analyse and evaluate such performance.

One example of these co-simulation approaches is a formal agent-based simulation framework (FABS) proposed by Niazi and Hussain (2011) in which the sensing algorithms as well as the environment e.g. playground specifying the node location, are implemented with Z formal specification. Netlogo, an agent-based simulation tool, is used to drive the simulation environment. The developed formal specification models (both the controller and environment) are generated into the Logo programming language which can be executed in this agent-based simulation environment. However, similar to other formal work described in the earlier section, this work does not focus on the proof of correctness of protocols and algorithms. The created formal specification is only a mathematical model that does not contain any invariant assertion (only contains type checking assertion e.g. the number of counted flocking boids in the immediate sensing radius must not be less than zero) indicating that the model behaviour still satisfies the system requirement including the system constraint and the safety property. Furthermore, the environment model driving in the simulation does not contain the realistic physical platform and environment as the WSN simulation normally has. Thus, this co-simulation framework is well away from the realism which is required by WSN development.

The closet to our co-simulation framework is CaVi (Boulis et al., 2008) that improved the performance of wireless networks accomplished by the WSN simulation with the formal performance analysis. This framework provides a uniform modelling language with a graphical user interface to design and implement the different network topology easily.

This results in a graphical style of specification model which is then generated to either a formal model in the PRISM model checker or an input model for the Castaslia simulator. Consequently, the benefits of model checking and simulation are combined. Not only the network performance based on the realistic channel/radio is analysed by the simulator but also more detailed analysis is taken into account by the probability of model checking in which the signal strength probability affecting the topologies in performance analysis is measured. This influences more robust performance measurement for simulation. However, this study only strengthens the network performance analysis by using the reception probability in model checking. As the behaviour of network depends on the protocol and algorithms, the correctness of communication protocols and algorithms is still required to verify by this framework. Furthermore, this framework does not support the component reusability as the PRISM model is tightly generated based on flooding and gossiping protocols. This limits the development of specific network protocols in this framework.

Consequently, this research contributes the integration of proof-based formal and WSN simulation approaches to create an alternative choice of formal co-simulation approaches to increase the quality of WSN development. This results in the supporting flexible and reusable co-simulation framework to be proposed. We facilitate proof mechanism provided by the proof-based formal method, Event-B, to encode and verify the communication protocols and algorithms in the executable formal controller model before they are deployed in the simulation environment. This also benefits the performance analysis of this executable controller model to be enhanced with more concrete and accurate environment models provided by WSN simulation, MiXiM.

3.13 WSN Deployment Problems

WSN deployment is concerned with embedding an operational sensor network into the real-world environment in order to monitor and control real-world phenomena. In many cases, the complexity of the environment may cause the system many problems which have not been detected during lab-based testing. As understanding the problems occurring during deployment is our first step, we reviewed twelve different projects which have different goals and requirements described in Beutel et al. (2010). Table 3.2 shows the characteristic of deployment selected by Beutel et al. (2010). The network size, hardware platform, the duration of deployment and network type were used as key features in selecting the project.

Figure 3.15 shows a fishbone diagram (Ishikawa, 1976) illustrating the causes of the problems that occur during fourteen different project deployments. These problems are classified into three classes: *node problems* regarding a single node, *network unreliability*

Table 3.2: Characteristics of selected deployment (Beutel et al., 2010)

Project	Year	#nodes	Hardware	Duration	Multi-hop
Great Duck Island I	2002	43	Mica2Dot	123 days	no
Great Duck Island II – patch A	2003	49	Mica2Dot	115 days	no
Great Duck Island II – patch B	2003	98	Mica2Dot	115 days	yes
Line in Sand	2003	90	Mica2	115 days	yes
Oceanography	2004	6	Custom HW	14 days	no
GlacesWeb	2004	8	Custom HW	365 days	no
Structural Health Monitoring	2004	10	Mica2	2 days	yes
Pipenet	2004/2005	3	Intel Mote	425-533 days	no
Redwoods	2005	33	Mica2Dot	44 days	yes
Potatoes	2005	97	TNode	21 days	yes
Volcano	2005	16	Tmote Sky	19 days	yes
Soil Ecology	2005	10	MicaZ	42 days	no
Luster	2006	10	MicaZ	42 days	no
Sensorscope	2006-2008	6-97	TinyNode	4-180 days	yes

problems regarding the communication reliability and *QoS problems* regarding Quality of Service in WSNs.

Considering **node death problems** which are the most common problems occurring during deployment, this causes a node to disappear from the network due to either power discharge or physical damage. Furthermore, low batteries can influence **wrong sensor readings**. In *Redwood Tree* project (Tolle et al., 2005), for example, the node which has a battery voltage lower than threshold recorded a high temperature above 40°C while other nodes read temperatures between 5°C and 25°C. This causes the node to report incorrect data. Considering **node death problems** which are the most common problems occurring during deployment, this causes a node to disappear from the network due to either power discharge or physical damage. Furthermore, low batteries can influence **wrong sensor readings**. In *Redwood Tree* project (Tolle et al., 2005), for example, the node which has a battery voltage lower than threshold recorded a high temperature above 40°C while other nodes read temperatures between 5°C and 25°C. This causes the node to report incorrect data.

Network unreliability is another common problem. As reported in a *Line in the Sand* project (Arora et al., 2004), dense sensor networks create network saturation and congestion. This causes a considerable amount of collision between packets which causes significant network unreliability. Inappropriate design of the MAC layer (Ramanathan et al., 2005) and repeated network floods in *LOFAR-agro deployment* (Langendoen et al., 2006) are other causes of traffic congestion that lead to multiple senders to transmit packets accidentally. This can potentially affect packet loss.

In addition to node death and network unreliability described above, they can influence **Quality of Service (QoS)**. Three categories of QoS problems are considered: **high report latency**, **low data yield** and **short network lifetime**. Firstly, **report latency**

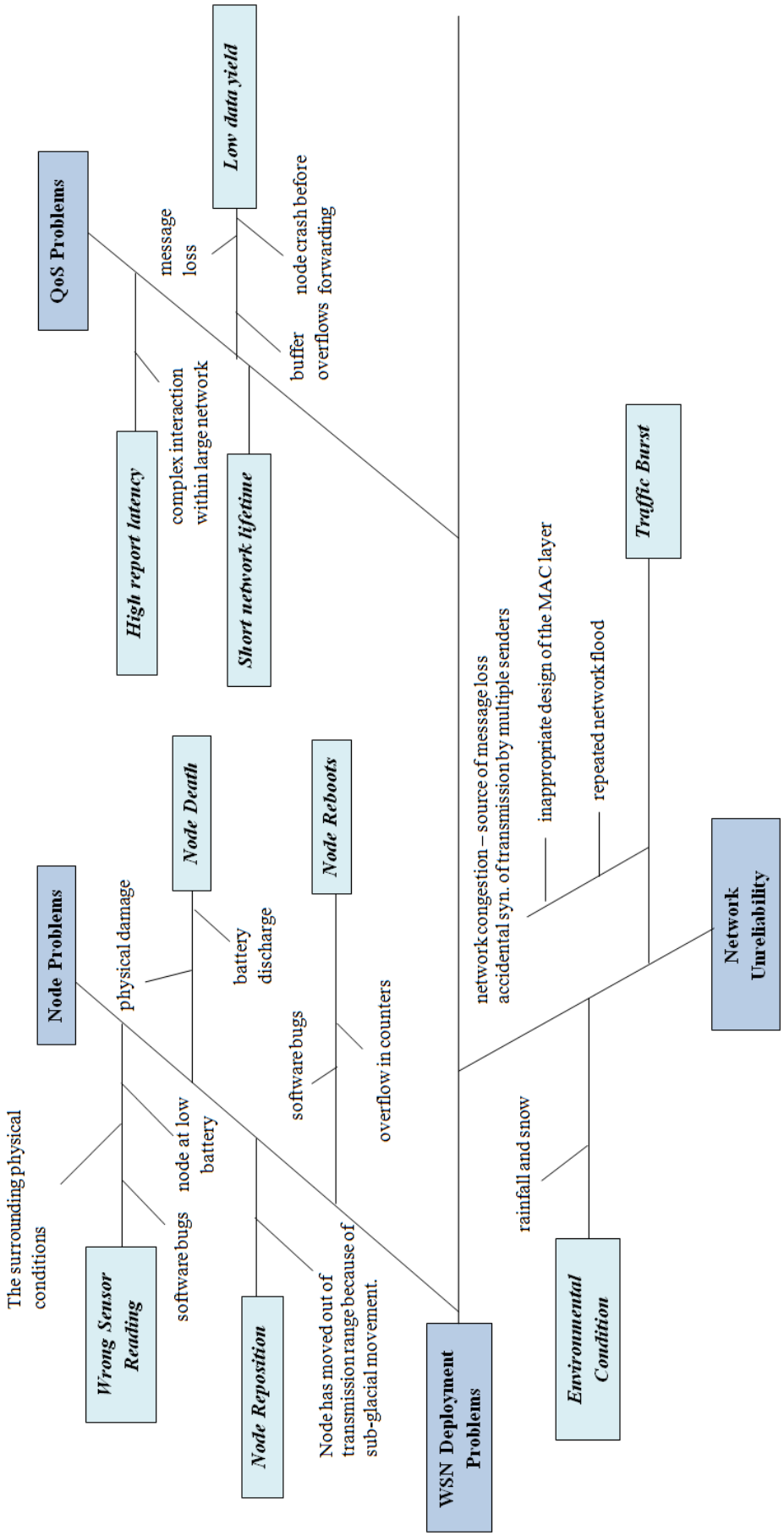


Figure 3.15: the causes of the problems occurred during twelve different project deployment

is a measure of the size of the time delay from the source node sensing and transmitting data to the sink receiving it. Thus, in WSN connection, **high report latency** causes the network connection to suffer from long delays. Secondly, **low data yield problems are related to packet loss problems**. For example, in the *Redwood Tree* deployment (Tolle et al., 2005), just only $20-30\%$ of data yield is reported due to packet loss caused from the problems discussed above. Finally, **short network lifetime problems** are caused by insufficient live nodes to report observations. Nodes in the network may not be able to stay longer for transmitting data to the sink due to energy depletion or complete damage.

Table 3.3 shows a summary of common deployment problems with their corresponding projects.

Table 3.3: Summary of common problems encountering during deployment with their corresponding projects

Project	Node problems				Management problem
	Node death			Wrong sensor reading	Last minute software update
	Physical damage	Power outage	Hardware Failure		
GDI I	✕			✕	
GDI II – patch A		✕			
GDI II – patch B		✕			
Line in Sand				✕	✕
Oceanography	✕				
GlacesWeb	✕				
SHM				✕	
Pipenet	✕	✕			
Redwoods		✕		✕	
Potatoes		✕			✕
Volcanos		✕			
Soil Ecology	✕				✕
Luster			✕		
Sensorscope			✕		

From this review, we summarise and prioritise common WSN development problems as follows:

(1) Connectivity problems caused by node death

(2) Network congestion caused by dense network

(3) Energy awareness or energy consumption issues

(4) Wrong sensor reading influence by energy depletion or hash environment

As the main objective of this research is to tackle the problems caused in the network level, we tackle issue (1) which are basic common WSN deployment problems.

3.14 Conclusion

In this chapter, the necessary WSN background was given. The current WSN practice was also discussed with the limitations that are going to be tackled in our research contributions that will be elaborated in Chapter 6. The applications and network protocols are typically co-designed and programmed very close to the specific OS and hardware for the specific motes. Therefore, programmers have to focus on the low-level system development e.g. embedded code based on specific platforms rather than high-level abstraction development e.g. in the design of network and communication algorithms. Furthermore, software code has to be integrated with APIs provided by OS or hardware platforms into a single image. This results in the software code to be very difficult to understand and maintain. Although the complexity of this monolithic development can be reduced by middleware, the focus of low-level system issues is still required by the development. Based on our literature reviews and surveys, we can conclude that the trends of WSN development improvement are the separation of concerns and the high level of abstraction.

Simulation-based development approaches have been considered to be used in WSN development process as they offer the high level of programming abstraction and effective testing together with performance analysis. Recently, various powerful simulators exist such as NS-2, OMNeT++, Castalia and MiXiM. The advantages provided by these simulators are, gaining better WSN understanding, designing network protocols and evaluating algorithm performance before deploying in the real world. Thus, a number of these simulators were discussed with their advantages against limitations in this chapter. We have discovered the limitation influenced by the simulation in which the testing technique enabled by simulation cannot guarantee the absence of certain classes of fault as formal methods do.

Since WSNs have been adopted in safety-critical domains, the highly reliable verification and validation techniques are needed by their development. Therefore, there are a number of adopted formal methods in WSN development. Executable formal methods have been introduced in the WSN development as the correctness of protocol algorithms needs to be guaranteed as well as the performance evaluation and analysis. However, the environment model used to execute the formal model is abstracted away from the real environment. As a result, this leads our thesis to contribute formal co-simulation framework for strengthening the WSN development by integrating the executable formal controller models in Event-B and more concrete and accurate environment models provided by WSN simulation, MiXiM. In order to demonstrate our contribution, this chapter also explored and prioritised the WSN problems occurring during deployment. In the next chapter, we will describe the WSN case study that is used to demonstrate the contribution. The details of simulation benefit and drawback will be demonstrated

and discussed in Chapter 5. This will illustrate how MiXiM can be used to tackle a basic common deployment problem that is the connectivity problem caused by node death.

Chapter 4

WSN Case Study Formulation

4.1 Introduction

In order to illustrate our core contribution, that is the integration of formal methods with WSN simulators, we formulate our WSN case study based on a real existing deployment, the SensorScope project (Barrenetxea et al., 2008a,b) aimed to provide an environmental monitoring system by means of WSNs as explained in Section 3.6.2. This deployment project motivates us to create our case study for three reasons. The first reason is that this deployment used a "*code-and-fix*" style of development process in which software code needs to be programmed in a low-level way. This limits the software to be co-designed and tested based on the operating system and hardware platform and causes the software to have high complexity and to be hard to maintain as discussed in Section 3.7. The other reason is that this project uses the standard WSN system architecture. This architecture provides the standard guidelines to implement WSN systems containing very well-known communication protocols and algorithms including MintRoute and duty cycle MAC protocols used by other deployment projects (Tolle et al., 2005; Werner-Allen et al., 2006). Finally, SensorScope faces node and thus link failure problems which cause packet loss. This fulfils one of these research focuses, that is to tackle the connectivity problem caused by node death.

As a result, our WSN case study is created based on the SensorScope project by replicating its network protocols and configuration. This case study provides an experimental test base which explores the strengths and weaknesses of traditional simulation (S-style) and formal (F-style) and our contribution: co-simulation (C-style) development for WSNs, proposed in research vision, Chapter 1. Section 4.2 provides an overview of this case study. Next, the requirements of the system are explained in Section 4.3. In Sections 4.4, 4.5 and 4.6, the SensorApp, MintRoute and Sensor-MAC protocols which are used as our application, routing and MAC protocols respectively, are described before presenting the non-functional requirements in Section 4.7. Section 4.8 describes

the fault scenarios of SensorScope project that motivate us to add fault injection to our approach. We provide the conclusion of this chapter in Section 4.9.

4.2 An Environmental Monitoring System

As mentioned earlier that our case study is created based on the SensorScope project, this section provides a description of an environmental monitoring system including its communication protocols and algorithms. In this system, environmental samples such as air temperature, wind speed and direction data are periodically sensed by each sensor before transmitting to a sink through multiple hops. In order to meet the requirements in multi-hop wireless networking, a communication stack for our case study has been designed. There are four layers as shown in Figure 4.1. However, this research focuses on the top three layers of the protocol stack which are Application, Network and MAC layers. Note that we do not include the transport layer in this stack. This is for two reasons. The first is our target simulator, MiXiM, provides the standard layers based on the IP models which does not include the transport layer in the protocol stack (Köpke et al., 2008). Secondly, in the SensorScope project, this transport layer is only responsible for creating packets with sensed data; including header fields: hop count and sequence number. In our simulation model, the packet is created in the application layer as soon as the node senses data. Furthermore, we implement reliable communication e.g. acknowledgement (ACK) mechanism at the MAC layer rather than at the transport layer.

In order to replicate our case study to be similar to the SensorScope project, *SensorApp*, a periodic sensing protocol in which the sensed data is transmitted periodically down to the lower layer, is chosen at the application layer. For the network layer, *MintRoute* (Woo et al., 2003), a link quality routing protocol, is selected to implement an efficient routing protocol. The duty cycling protocol was selected to express the medium access control (MAC) protocol in order to increase the nodes' lifetime. We used Sensor-MAC (S-MAC) (Ye et al., 2002) to represent this duty cycling protocol.

In addition, for the packet transmission mechanism, we implement two different kinds of packet:

- **Data packets:** these packets contain sensed data that must be routed periodically from a reading sensor to a sink. In this case study, these data packet are created by SensorApp protocol.
- **Control packets:** these packets are intended for a specific purpose regarding network control and maintenance. They can be a specific packet transmitted to a neighbour node or broadcast to all of them. In our case study, beacon and route update packets are defined in MintRoute in order to maintain the route tree (see

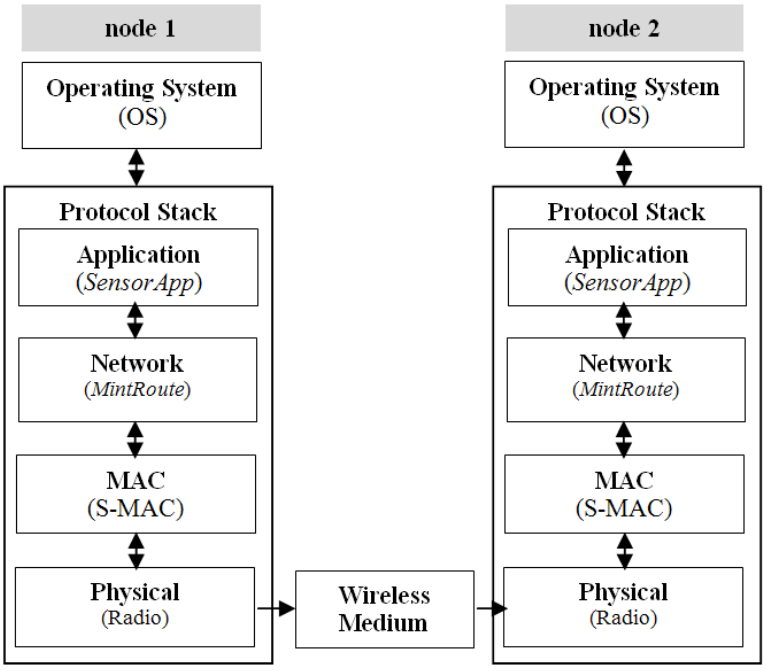


Figure 4.1: Communication stack of our case study

Section 4.5 for more details). For the S-MAC protocol, synchronisation packets (SYN), request-to-send (RTS), clear-to-send (CTS) and acknowledgement (ACK) packets are defined for clock synchronisation and reliable communication at MAC layer respectively (see Section 4.6 for more details).

4.3 System Requirements

In this section, we outline the user requirements derived from the SensorScope project. We also structure the requirements by using various labels in order to trace the completeness of the system easily. The requirement labelling guideline identified in Abrial (2010) are applied to requirements and assumptions of this case study. Here the labels used in this case study are:

- **FUN** for functional requirements
- **REL** for reliability requirements
- **PER** for performance requirements
- **ENV** for environment assumptions

Tables 4.1 and 4.2 provide an overview of the categorised functional and non-functional requirements respectively. A detailed description of these requirements is presented in

the following sections. Consequently, Sections 4.4, 4.5 and 4.6 describe the detailed description of functional requirements of the SensorApp, MintRoute and S-MAC protocols respectively, whereas the detailed non-functional requirements are described in Section 4.7.

Table 4.1: An overview of the structured functional requirements of system case study

ID	Requirements
SensorApp protocol	
FUN-1	Sensor nodes' functionality
FUN-2	Sink's functionality
MintRoute protocol	
FUN-3a	Neighbourhood discovery
FUN-3b	The Link Quality Estimation
FUN-3c	Route Broadcast Mechanism
FUN-3d	Parent Selection
FUN-3e	Data Packet Forwarding Mechanism
FUN-3f	Failure Detection Mechanism
S-MAC protocol	
FUN-4a	Duty Cycling of S-MAC
FUN-4b	Choosing and Maintaining Schedules
FUN-4c	Collision Avoidance

Table 4.2: An overview of the structured non-functional requirements of system case study

ID	Requirements
Network and link assumptions	
ENV-1	The dynamic link connectivity assumption
ENV-2	The packet transmission assumption
Reliability and fault tolerance requirements	
REL-1	The dynamic high-quality route requirements
REL-2	Acknowledgement mechanisms for end-to-end reliability of transmission
Performance requirements	
PER-1a	Network load distribution analysis
PER-1b	Network latency analysis
PER-2	The number of packet loss analysis

4.4 SensorApp Concepts

SensorApp is a periodic sensing protocol at the application layer in which the sensed packet is transmitted periodically down to the lower layer. Here are the functional requirements for this application protocol.

- **Sensor nodes:**
  - **FUN-1:** each sensor node has two different roles: (a) data reader and (b) data router.
    - **FUN-1a:** each node senses data periodically and stores its sensed data in a sense buffer. Then, such data in the sense buffer is transmitted directly to its neighbour. After the sensed data is transmitted, it is deleted from sense buffer.
    - **FUN-1b:** the receiving sensor node records such packet into its receive buffer before forwarding this packet to the neighbouring node. Then, it clears the receive buffer after it transmits the packet to its neighbouring node.
- **A Sink:**
  - **FUN-2**: A sink collects data from sensors. It has its own buffer for storing data. After the data is transmitted to a database server, it clears data from its buffer.

4.5 MintRoute Concepts

MintRoute (Woo et al., 2003) is a routing algorithm to build the route tree from every node towards a sink. The route tree is dynamically changed based on the link quality between nodes. This link quality is adjusted by the successful rate of transmitted packet delivery. MintRoute performs four major steps: *neighbourhood discovery*, *the link quality estimation*, *route broadcast* and *parent selection*.

4.5.1 FUN-3a: Neighbourhood Discovery

Each node discovers its neighbours from neighbours' periodically rebroadcasting beacon packets initiated by a sink. It records information about the nodes described in such beacon packets as a data entry in the *neighbour table*. Figure 4.2 shows the structure of neighbour table.

Id	Parent	cost	Missed	Received	lastSeqno	...	receiveEst	sentEst

Figure 4.2: Neighbour table structure

where

- *id* represents neighbour address.
- *parent* represents neighbour's parent.
- *cost* represents neighbour's path cost to a sink (via its parent). In this case study, the path cost of neighbour is calculated from the summation of link quality ratio towards the sink of that neighbour and path cost of its parent. For example, the path cost of neighbour node j ($Path\ Cost_j$) can be calculated as expressed in Figure 4.11.
- *missed* represents the number of lost packets (within the link quality estimation interval) from this neighbour.
- *received* represents the number of received packets (within the link quality estimation interval) from this neighbour.
- *lastSeqno* represents the sequence number of the most recent packet transmitted from this neighbour.
- *receiveEst* represents the estimation of reception link quality (from the neighbour to this node) as shown in Figure 4.5.
- *sentEst* represents the estimation of transmission link quality (from this node to the neighbour).

In the initial process of neighbourhood discovery, a sink broadcasts "*a beacon packet*" to its neighbours periodically. Then, each receiving node starts rebroadcasting such a received beacon to its neighbours. This rebroadcasting process is repeated until every node in the network has received the rebroadcast beacon packet. Figure 4.3 shows neighbourhood discovery steps. A sink initiates broadcasting a beacon packet to nodes 2 and 3 which are its neighbour nodes as shown in Figure 4.3(b). Nodes 2 and 3 discover the sink as their neighbour and record the information of the packet (such as *missed*, *received* and *lastSeqno*) into their neighbour table. Then, both nodes rebroadcast this received beacon message to their neighbours (in Figure 4.3(c)). Node 1 who is one neighbour of both nodes, for example, can discover nodes 2 and 3 from its received beacon packets to be its neighbour by recording their information into the neighbour table as shown in Figure 4.3(c). Figure 4.4 shows the pseudocode corresponding to the neighbourhood discovery steps described above.

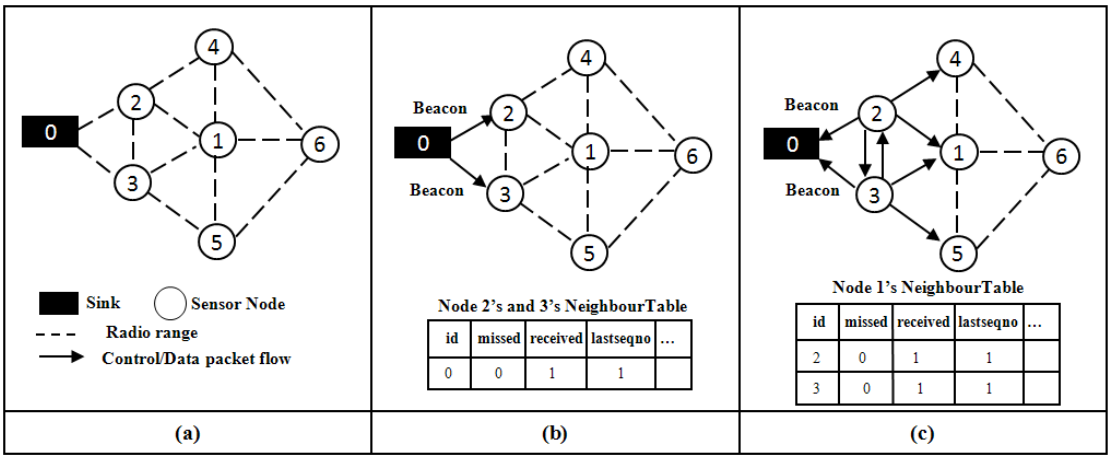


Figure 4.3: Example of neighbourhood discovery mechanism

Algorithm *NeighbourhoodDiscovery*
when
a node receives a beacon packet.
then
if the source does not exist in the neighbour table **then**
add the source as a new entry in the neighbour table.
end
calculate the number of missed packet and the number of received packet from that neighbour.
record the sequence number, the number of missed packet and the number of received packet corresponding to such source into the neighbour table.
end

Figure 4.4: Pseudocode *NeighbourhoodDiscovery*

4.5.2 FUN-3b: Link Quality Estimation

Each node estimates periodically the link quality of its neighbours by observing the successful rate of transmitted beacon packets. The MintRoute protocol uses two kinds of link quality for both directions, *reception (inbound)* and *transmission (outbound)*, to choose a parent of each node. The reception link quality is firstly determined as *packet reception ratio (PR)*. Then, this ratio is used as the transmission link quality by the neighbouring node. The pseudocode of *LinkQualityEstimation* is shown in Figure 4.5.

In order to estimate the link quality periodically, the timer indicating the estimation interval is identified e.g. link quality estimation is performed every period t as can be seen in Figure 4.5. When such a timer reaches its repeated interval, each node calculates the number of received beacons from each neighbour in time interval *(t-1, t)*. For example, if a beacon is broadcast periodically from a sink every 120s and the reception estimation ratio requires to be calculated every 5 received beacon, the time interval for this link quality estimation will be set to be after every 600s. This number is divided with the number of actual transmitted beacons in the same interval in order to calculate the reception link quality (*receiveEst*) of each neighbour as shown in the equation in Figure 4.5. Figure 4.6 shows an example of link quality estimation performed

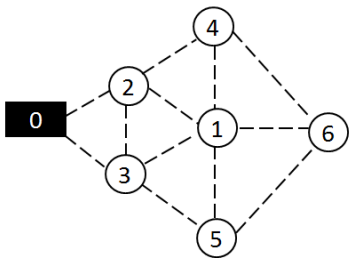
Algorithm *LinkQualityEstimation*
when
timer interrupt occurs.
then
for each entry *j* **in** the neighbour table **do**
calculate the packet reception ratio (*receivedEst*) as

$$PR_j\,(t-1,t) = \frac{the\ number\ of\ received\ beacons\ in\ (t-1,t) from\ j}{the\ number\ of\ actual\ transmitted\ beacons\ in\ (t-1,t)} \times 100$$

record the PR_j to be *receivedEst* into the neighbour table.
end

Figure 4.5: Pseudocode *LinkQualityEstimation*

at node 1. In this example, we assume that each node actually transmits five beacons during periodic *(t-1,t)*. However, node 1 receives four and three beacons from nodes 2 and 3 respectively. Therefore, the reception link quality of nodes 2 and 3 is 80% and 60% respectively as shown in Figure 4.6.



Node 1's NeighbourTable

id	missed	received	lastseqno	receivedEst	...
2	1	4	5	80	
3	2	3	5	60	
...					

Figure 4.6: Example of link quality estimation

4.5.3 FUN-3c: Route Broadcast Mechanism

After each node estimates its reception link quality, it broadcasts "*route update packets*" periodically to its neighbours. The route update packet includes parent address, estimated routing path cost to the sink and a list of reception link estimations of each neighbours. The fields of a route update packet is shown in Figure 4.7.

parent	cost	estEntries	estList

Figure 4.7: route update packet Structure

where

- *parent* represents this node's parent.
- *cost* represents this node's cost to a sink (via his parent).
- *estEntries* and *estList* represent the neighbour node id and its corresponding reception link quality ratio (*receiveEst*) respectively.

When the receiving node receives this route update packet from the node identified in the neighbour table, it updates the corresponding entry with the value attached in the packet. Especially, *the reception link quality (receiveEst)* attached in field *estList* is considered to be transmission link quality and updated into field *sentEst* in the neighbour table.

For example, we assume that the *receiveEst* of nodes 2 and 3 in node 1's neighbour table to be 80% and 60% respectively as shown in Figure 4.8(a). These ratios are attached in field *estList* of route update packet created by node 1 as illustrated in Figure 4.8(b) before this packet is broadcast to its neighbours. Note that in the first period of broadcasting route, each node has not discovered its parent. As a consequence, parent and path cost of parent in this route broadcast packet are attached with NULL and zero. When node 2 which is one of node 1's neighbours receives this packet, it records *receiveEst* from such packet to be its *sentEst* which is 80% as shown in Figure 4.8(c).

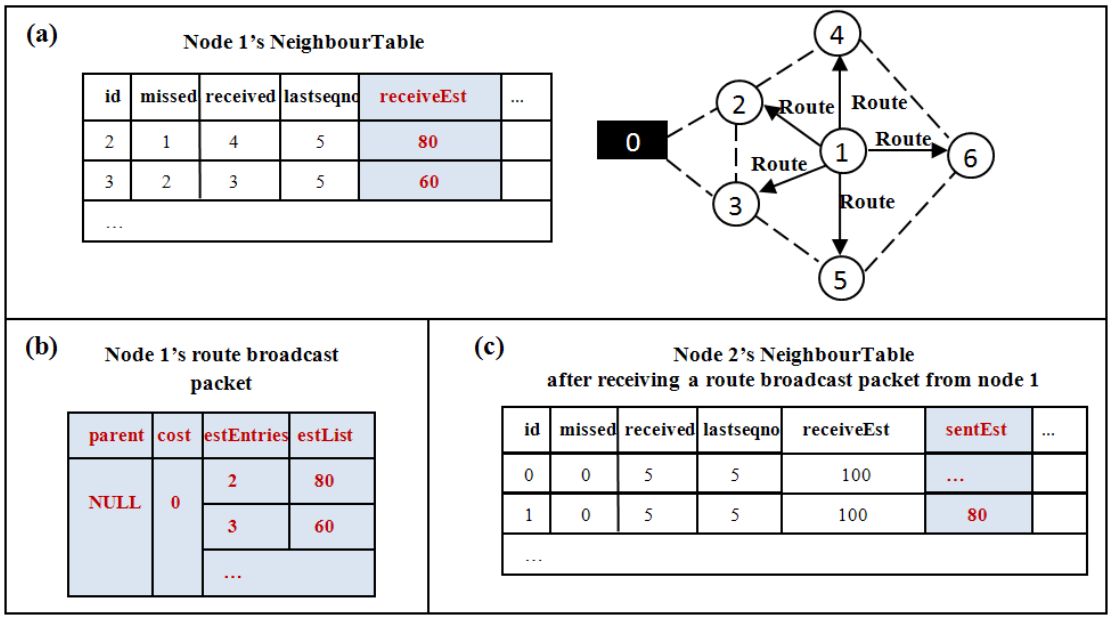


Figure 4.8: Example of route broadcast mechanism

Figure 4.9 and 4.10 show the pseudocode for broadcasting and receiving route update packet corresponding to the mechanisms mentioned above.

```
Algorithm BroadcastRouteUpdatePacket
when
  timer interrupt occurs.
then
 attach a current parent and cost for such current parent to a sink to a route update packet.
 for each entry j in the neighbour table do
    attach neighbours' reception ratio of entry j to the route update packet.
 broadcast the route update packet to the neighbours.
end
```

Figure 4.9: Pseudocode *BroadcastRouteUpdateMessage*

```
Algorithm ReceiveRouteUpdatePacket
when
  a node receives a route update packet.
then
 record the corresponding attached current parent, cost of such parent and packet reception ratio
 (receivedEst) as the transmission link quality (sentEst) into the neighbour table.
end
```

Figure 4.10: Pseudocode *ReceivesRouteUpdateMessage*

4.5.4 FUN-3d: Parent Selection

A neighbour with the smallest *Path Cost* is chosen as a parent as illustrated in Figure 4.11. Furthermore, the link which has the reception and transmission link quality less than the quality threshold is not considered to be a parent. Figure 4.12 shows an example of parent selection of node 1. Figure 4.12(a) illustrates the possible *receiveEst* and *sentEst* between node 1 and its neighbours. Considering Figure 4.12(b), links from node 1 to nodes 3 and 5 are excluded due to their link quality being less than the link quality threshold which is set to 80%. Figure 4.12(c) shows the process after the parent of node 1 is chosen. As MintRoute protocol chooses a parent from the node which has the high link quality towards a sink, node 6 is not chosen to be a parent of node 1, even though it has the highest link quality compared to other neighbor nodes. This is because it is furthest away from the sink. Consequently, it has the highest path cost compared to other nodes which have a lower link quality. Node 2 that has the next highest link quality but a lower path cost is chosen as a parent of node 1.

For the next route broadcast interval after the parent of each node is selected, this parent, estimated routing path cost towards a sink and the new reception link estimation of neighbours are attached in the route broadcast packet as shown in Figure 4.13(a). When the neighbouring nodes receive this packet, the entries attached in such a packet are updated into the corresponding entry in the neighbour table as shown in Figure 4.13(b). Note that x indicates estimated routing path cost from node 1 towards a sink.

Algorithm *ChooseParent*
when
timer interrupt occurs.
then
select a parent(P) as

$$P = \min_{j \in NB} (Path\ Cost_j)$$

where

$$Path\ Cost_j = \left(\frac{100}{receiveEst_j} \times \frac{100}{sentEst_j} \right) + Path\ Cost\ of\ Parent_j$$

end

Figure 4.11: Pseudocode *ChooseParent*

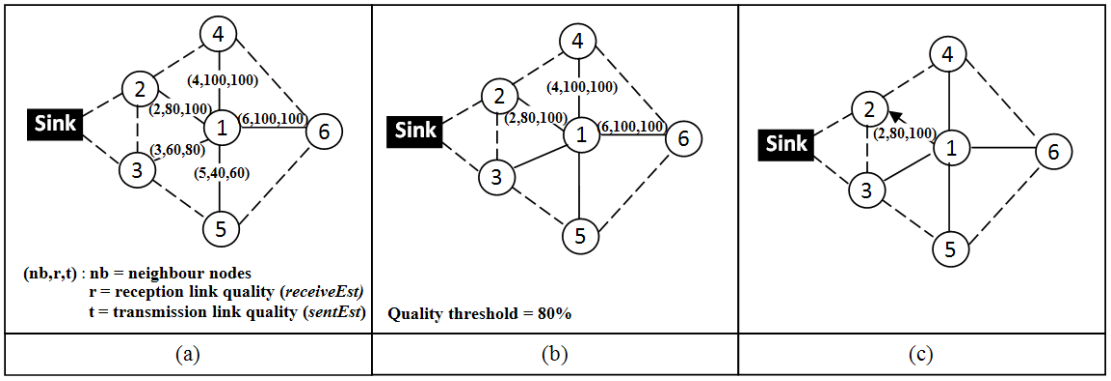


Figure 4.12: Example of parent selection

4.5.5 FUN-3e: Data Packet Forwarding Mechanism

As the network layer is responsible for packet forwarding from one node to neighbour node(s), MintRoute provides two types of data packet forwarding mechanisms: (1) unicast and (2) broadcast data packet forwarding mechanisms.

1. **Unicast data packet forwarding mechanisms:** this mechanism basically relays a data packet from one sender/forwarder to a specific neighbour node. In MintRoute protocol, this specific neighbour node is identified from the route tree.

2. **Broadcast data packet forwarding mechanisms:** the mechanism enables the data packet to be relayed from one sender/forwarder to all neighbour nodes. MintRoute implements this mechanism by checking that if the route tree has been constructed, the transmitted data packet will be broadcast to all neighbour nodes of sender/forwarder.

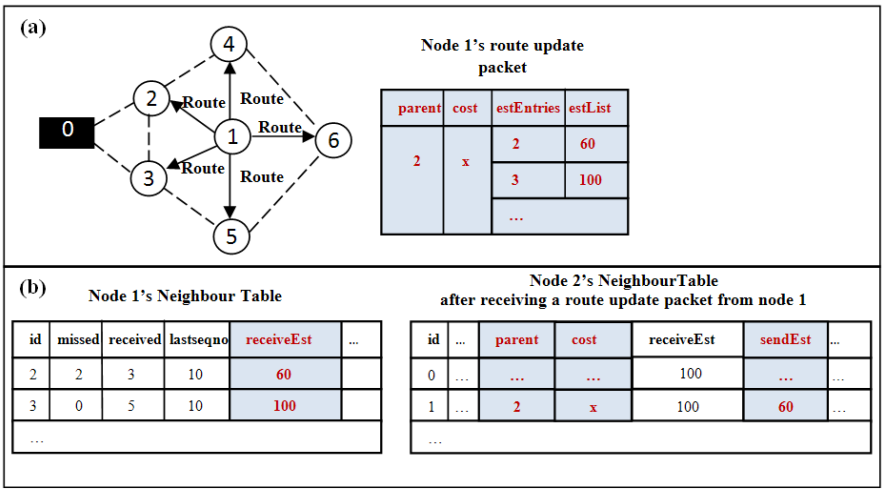


Figure 4.13: Example of broadcasting route update message after a parent is selected

4.5.6 FUN-3f: Failure Detection Mechanism

MintRoute provides the poor/dead neighbour detection mechanism in which each neighbour in the neighbour table is maintained by the liveliness value. This liveliness is set at a high value when a new neighbour node is discovered. This value decreases every time when the sensor node estimates the link quality of its neighbours and is reset back when the sensor node receives a route update packet from the neighbour. Dead neighbours are identified in the neighbour table when their liveliness value becomes zero.

4.6 S-MAC Concepts

S-MAC (Ye et al., 2002) is a duty cycling protocol modified from contention based medium access protocol IEEE 802.11. It is a synchronous MAC protocol which manages the synchronization and periodic sleep-listen schedules locally.

4.6.1 FUN-4a: Duty Cycling of S-MAC

In this medium access control protocol, time for communication in S-MAC is divided into frames. Each time frame has two parts: *listen period* and *sleep period*. By using S-MAC, each node periodically schedules the fixed listen/sleep cycle as shown in Figure 4.14.

In principle, the radio of each node has two statuses: on and off. The node can transmit(TX)/receive(RX) data only when its radio is turned on. During Sleep period, the node turns off its radio in order to preserve energy. The node turns on its radio during a listen period in order to communicate with its neighbours and send any control

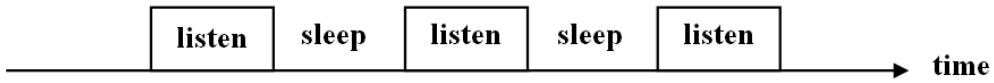


Figure 4.14: The S-MAC duty cycle

packets. This period is divided in two periods: timer synchronisation (SYNC) and data transmission (DATA) periods (Canli et al., 2010) that are described in Sections 4.6.2 and 4.6.3. As one of the major sources of energy waste is from the listen period, it must be a small percentage of the active duty cycle. This means that the sleep period must be longer than the listen period for energy-efficiency purposes.

4.6.2 FUN-4b: Choosing and Maintaining Schedules

As S-MAC is a synchronous protocol in which all nodes have to turn on their radio at the same time, each node needs to specify its schedule and exchange this schedule with its neighbours. The schedules of all known neighbours are recorded in a schedule table. Here are the steps to choose the schedule and create their schedule table.

1. The node starts with listening within a certain amount of time. If the node does not receive a schedule from another node, it randomly selects a time to go to sleep. Then, its schedule attached in a SYNC message is broadcast immediately to indicate the time when it will go to sleep.

2. Whenever the node receives a schedule before choosing its own schedule, it follows that schedule by setting its schedule to be the same as the received schedule. Then, it rebroadcasts this schedule to its neighbours, indicating the time it will sleep.

4.6.3 FUN-4c: Collision Avoidance

RTS/CTS (*Request to send/clear to send*), other control packets, are used as a handshaking between two sensor nodes before sending a data packet. This mechanism is also used to avoid collisions between multiple transmitting nodes. Figure 4.15 illustrates the basic S-MAC operation illustrating sender node 2 has data to be sent to destination node 0. This starts when node 2 creates request-to-send (RTS) packet and transmits it to node 0. When node 0 receives RTS packet from node 2, it creates clear-to-send (CTS) packet and transmits it back to node 2. After node 2 receives node 0's CTS packet, it transmits a data packet to node 0. Finally, node 0 returns acknowledgement (ACK) back to node 2 after it receives the transmitted data packet.

As in IEEE 802.11, each node in S-MAC maintains a Network Allocation Vector (NAV) for virtual carrier sensing. For example, node 3 in Figure 4.15 is a neighbour of node 2

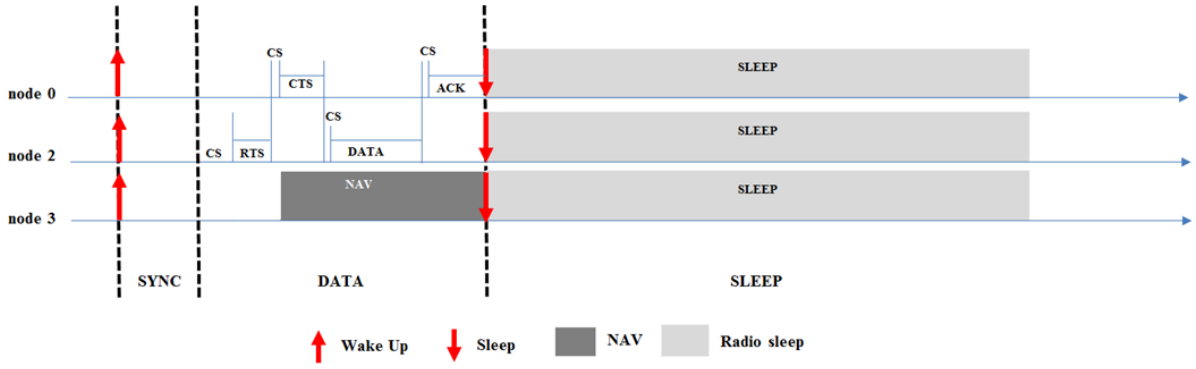


Figure 4.15: Collision avoidance mechanism in S-MAC

and overhears the RTS and CTS sent by node 2. Node 3 will set its NAV to indicate this virtual carrier and will not send any traffic while its NAV is nonzero.

Carrier sense (CS) delay is also used in order to eliminate collision when multiple nodes try to send data in the same DATA period. This is accomplished by introducing the random contention delay (a random amount of time) to such simultaneous nodes. Only one node allocating the shortest contention delay can access a channel to transmit its data packet. The other nodes need to wait for the next contention (when this packet transmission is finished).

4.7 Non-Functional Requirements

This section describes the detailed description of non-functional requirements of our case study. We classify these non-functional requirements into three categories: *network and link assumptions*, *reliability and fault tolerance*, and *performance requirements*.

Network and link assumptions:

- **ENV-1:** nodes are connected by the dynamic links in which their quality may change at any time.
- **ENV-2:** a data packet must be either transmitted successfully or lost from a source node to a sink in the dynamic network.

Reliability and fault tolerance requirements:

- **REL-1:** the dynamic routing protocol should provide the reliability and fault tolerance of link connectivity in a route tree for data transmission:
  - **REL-1a:** to achieve the reliability, the dynamic routing protocol should find the best available high-quality route for data transfer. Thus, the routing protocol should record link connectivity statistics recorded dynamically and construct such a high-quality route based on such connectivity statistics. This reliability mechanism is provided in MintRoute as already mentioned in Section 4.5.
  - **REL-1b:** the routing protocol should provide the disconnected/dead link detection mechanism so that the route tree must be recovered after the disconnected/dead link happens in the route tree. This fault detection and tolerant mechanism is provided in MintRoute as already mentioned in Section 4.5.6.
- **REL-2:** in order to improve the per-hop and end-to-end reliability of transmission, the primitive implicit acknowledgement is used as already mentioned in S-MAC, Section 4.6. Whenever a node forwards a message to its parent in the route tree, it will receive acknowledgement sent from its parent eventually. if the node does not receive the acknowledgement, it will assume that its parent has not received the message. In this case, it will retransmit the packet again.

Performance requirements:

- **PER-1**: the system must allow the clients to measure **PER-1a**: load distribution of network and **PER-1b**: latency.
- **PER-2**: the system must enable the clients to be able to measure and analyse the number of packet loss.

4.8 Fault Scenarios

The SensorScope project was conducted in six deployments from inside campus to the high mountain sites, containing a range in size between 6 to 97 stations (Barrenetxea et al., 2008b). The Génépi rock glacier deployment in Switzerland was one of the most challenging and successful deployments (Barrenetxea et al., 2008a). Before this project was deployed, the network algorithms were tested on an indoor testbed placed in their offices. The indoor experiment showed clearly that all the data from all sensor stations was able to be collected. However, during the outdoor deployment, hardware failures occurred to some sensor nodes which caused packet loss. Hence, this failure will be replicated into our simulation by injecting node failure to test the system.

4.9 Conclusion

This chapter has introduced a WSN case study aiming to investigate and evaluate the strengths and weaknesses of different WSN development styles including traditional simulation-based (S-) and formal-based (F-) and our contribution, formal co-simulation (C-style) development proposed in our research vision, Chapter 1. As discussed in the previous chapter, we have discovered that SensorScope deployment provides an experimental test base built on the contribution of C-style and integrated working environment. As this project is one of the current deployment projects in which the development and testing of applications and network protocols have to be closely tied up with the specific OS and hardware for the specific sensor node, it requires a high level of abstraction and separation of concern approaches and techniques to manage this complexity. Furthermore, SensorScope experienced failure after deployment. Thus, powerful V&V is needed by the development of this project. Consequently, our case study is derived for this real deployment project in order to represent the current practice in WSN development. MintRoute has been chosen to implement an efficient routing protocol and S-MAC has been used to represent the duty cycling MAC protocol. The configurations of these protocols in SensorScope are also replicated for this case study.

In order to demonstrate our research contribution, the next chapter will highlight how the traditional S-style development can benefit the current practice WSN development in terms of high level of abstraction and the performance analysis by using this case study. This case study also influences us to demonstrate how proof-based formal methods like Event-B can add value to the current practice of WSN development that will be described in Chapter 6. Finally, we will use this case study to demonstrate our core contribution C-style development which combine the strengths of both styles together as explained in Chapter 7.

Chapter 5

Simulation Development (S-style Modelling)

5.1 Introduction

To explore the benefits and drawbacks of S-style development in order to support sub-contribution C.1 in Chapter 1, this chapter describes the MiXiM simulation experiment on the environmental monitoring case study with the replicated network protocols and configuration from the real existing deployment, namely the SensorScope project as described in the previous chapter. In Section 5.2, the simulation model development for the upper layer protocol, SensorApp and MintRoute is demonstrated. Since one focus of our research is packet loss, Section 5.3 gives an overview of MiXiMs physical and environment models, in particular covering lossy communication. Section 5.4 describes the choice of simulation configuration data. In outdoor deployment, SensorScope experiences node and thus link failures which causes packet loss. We have replicated these problems by injecting node failure into our simulations. In section 5.5, we discuss the simulation results of two experiments with these failures. The conclusion of this chapter is described in Section 5.6.

5.2 Simulation Model Development

The component architecture for the development environment provided by MiXiM layers the development environment into the standard IP protocol stack as shown in Figure 4.1. Each layer can communicate with the adjacent layer via communication interfaces named gates. In our simulation model, we extended the base modules (the general structure) for application and network layers provided by MiXiM to implement SensorApp and MintRoute respectively. The existing code for periodic sensing protocol

provided by MiXiM was reused to implement SensorApp. We applied the MintRoute source code provided by tinyos-1.x[1] to develop our own routing algorithm. We also implemented the S-MAC protocol at MAC layer. In order to enable each layer model to communicate to the adjacent layer model, the standard interfaces provided by simulation toolkit are integrated into each single layer model. These standard interfaces provide the sending/receiving packet mechanism between each layer. They also provide data transmission at the different layer of a sender downwards and destination upwards by implementing packet construction/encapsulation and decapsulation at each layer. This enables the packet to be encapsulated incrementally at a sender downwards through application, network, MAC and physical layers respectively and decapsulated decrementally at a destination upwards. Finally, these protocol models were integrated with MiXiM's existing physical, wireless channel and battery models.

5.3 Lossy Communication

As the contribution of this research will be demonstrated by tackling the packet loss problem, two possible sources of problems have been considered in our simulation experiment. The former is *the node failure injection* in which we integrate our implemented node failure models into our simulation models in order to identify the specific node death in the network. Whereas the latter is *lossy communication* in which we apply the effect of interference wireless communication, received signal strength, signal-to-interference-plus-noise ratio (SINR) measurement facilitated by the channel, radio and analogue models provided by MiXiM. In this section, we provide a brief description of these mechanisms.

5.3.1 Interference Wireless Communication

As mentioned already in Section 3.9.2.4, the connection between nodes is indicated by the maximal interference distance. The maximal interference distance is a conservative boundary indicating the communication range based on the maximal distance between nodes. This means that a node can connect with another node when they are placed within the maximal interference distance. This interference may affect a node to possibly create ambient noise to disturb the communication of a neighbour. Packet collisions happen, when multiple interfering senders simultaneously transmit their packet to the same receiver. This leads to packet loss. For the unicast packet forwarding mechanism, this collision problem can be avoided by the MAC protocol e.g. S-MAC protocol by using the RTS/CTS and acknowledgement mechanism. However, this problem sill occurs in the multicast and broadcast mechanism as it does not facilitate the RTS/CTS handshake mechanism for reliable transmission due to the higher communication traffic and

[1]see MintRoute source code in tinyos-1.x http://tinyos-1.net/tos/lib/MintRoute

overhead. As Mintroute protocol builds the dynamic route tree based on the successful rate of beacon broadcast transmission, packet collision may occur when the beacons are broadcast from multiple senders to the same receiver at the same time. Therefore, we use the effect of packet collision caused by interference in the communication among nodes to create random packet loss mechanism. This validates our implemented MintRoute protocol by degrading the link quality between interfering nodes.

5.3.2 Received Signal Strength

The communication between nodes depends not only on a physical distance between the nodes but also the signal strength. The signal strength of a message sent from one node to another is affected by the environment that it travels through as shown in Figure 5.1. Once a packet is transmitted by the sending MAC protocol to its physical layer, the physical layer, then, converts that packet into a signal with its transmission (Tx) power calculation before transmitting it via the wireless channel. The signal is attenuated by various causes such as fading[2], shadowing[3]and path loss[4]as well as interfering transmitters while it is transmitting over the channel. Finally, this distorted signal is received at physical layer of the receiver and reassembled as a packet passed to the receiving MAC layer.

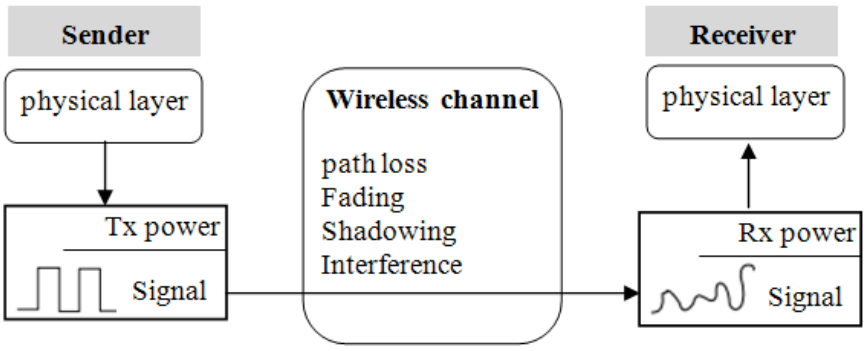


Figure 5.1: Aspects of a signal transmission(Wessel et al., 2009)

In MiXiM, the strength of a received radio signal is calculated by the *Decider* model interacting with the *Analogue* model provided by MiXiM. This analogue model is responsible for calculating the attenuation of a signal from the fading, shadowing and path-loss models. This attenuation calculation ($f_{att_{1..n}}$) is performed at the start of the reception of an arriving message. In MiXiM, the attenuation is represented as a negative value of decibels (dB) in order to reduce the received signal strength as shown in Figure

[2]Fading is deviation of the attenuation affecting a signal over certain propagation media.

[3]Shadowing is the effect that the received signal power fluctuates due to objects obstructing the propagation path between transmitter and receiver.

[4]Path loss is the reduction in power density (attenuation) of an electromagnetic wave as it propagates through space.

5.2. This calculation will then be considered together with the power of transmission (f_{TX}) to calculate the receiving power (f_{RX}) by the Decider model as shown in (5.1).

$$f_{RX} = f_{TX} \times \prod f_{att_{1..n}} \tag{5.1}$$

Where

f_{RX} denotes the receiving power of the signal (mW)

f_{TX} denotes the transmitting power of the signal (mW)

$f_{att_{1..n}}$ denotes the attenuation factor 1 to 10 (dB)

Note that, in MiXiM, the mathematical mapping function is used to express the aspects of simulating the wireless transmission process. For example, the signal sent to the channel might consist of a mapping for the power $f_{TX} : time \times frequency \times space \rightarrow power$ (Wessel et al., 2009). This mapping function is also used to map the receiving power as $f_{RX} : time \times frequency \times space \rightarrow power$ and attenuation as $f_{att_{1..n}} : time \times frequency \times space \rightarrow attenuation$.

In order to gain greater understanding of the receiving power calculation expressed in (5.1), Figure 5.2 demonstrates how the attenuation reduces the signal strength during transmission. This figure demonstrates the situation of the shadowing attenuation that affects the received signal strength from an obstructing object in the propagation environment. It can be seen that an obstructing object a between the communication of node s and z results in a weak received signal compared to that of a non-obstructed communication between node s and y that has the same distance. In shadowing situation expressed in Figure 5.2, we assume that object a causes -20 dB of attenuation which is converted to 0.01 mW. This affects the received signal strength to be reduced by multiplying the transmitted power with 0.01 mW of attenuation as shown in (5.1).

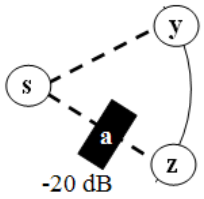


Figure 5.2: Example of attenuation reducing the signal strength (Wessel et al., 2009)

This receiving power can be used to determine the strength of signal in each *airframe*, a message at the physical layer sent across the channel. Note that one airframe is an encapsulated packet from upper layer in MiXiM. The busy-threshold is used to compare with this value to indicate whether the signal is weak. If the calculated receiving power

is less than the busy-threshold, this means that the signal of the airframe is weak leading the receiver to discard that airframe.

5.3.3 Signal-to-Interference-plus-Noise Ratio (SINR)

After the decider calculates the receiving power of an incoming airframe and indicates that it has sufficient signal strength to be received, the *signal-to-interference-plus-noise ratio*(*SINR*) is measured. The SINR of received signal is measured in order to decide whether the transmitted message is noise or receivable. In MiXiM, this mechanism is integrated into Decider models. The decider will calculate the SINR of the received frame by using the division of the receiving power of the signal (f_{RX_S}) calculated based on (5.1) with the summation of the receiving power of every interfering signal (Wessel et al., 2009) as shown in (5.2). This f_{SINR} is compared with the SINR-threshold to decide whether the received frame will be sent up. If this value is less than this threshold, the received frame will be treated as a noise by dropping that packet.

$$f_{SINR} = \frac{f_{RX_S}}{\sum f_{RX_{noise_{1..n}}}} \tag{5.2}$$

Where

f_{SINR} denotes the signal-to-interference-plus-noise ratio (mW)

f_{RX} denotes the receiving power of the signal of an incoming airframe (mW)

$f_{RX_{noise1..n}}$ denotes the noise calculated from the summation of the receiving power of all interferencing airframe (mW)

An example of SINR calculation has been shown in Figure 5.3. This figure demonstrates the scenario in which the physical layer of nodes 2, 4 and 5 transmit airframe X, Y and Z respectively to the same node that is node 1. Assuming that the current received airframe is airframe Y. Airframes X and Z arrived in port concurrently with airframe Y. In this example, the SINR of incoming airframe Y is calculated at time *t4* (the end of receiving process). Thus, the receiving power of interfering airframes X and Z is treated as a noise of airframe Y. As a result, the SINR of airframe X can be calculated as illustrated in (5.3).

$$f_{SINR_Y} = \frac{f_{RX_Y}}{f_{RX_X} + f_{RX_Z}} \tag{5.3}$$

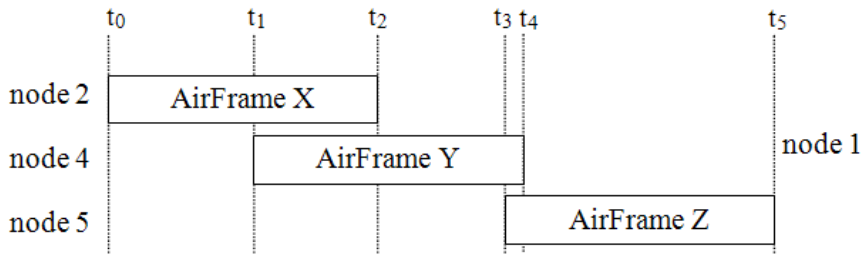


Figure 5.3: Airframes in the channel

5.4 Configuration Parameters

The real configuration parameters used in Barrenetxea et al. (2008b) were replicated for our simulation experiments. This included a sampling time interval of 120 seconds for the application protocol and parent selection, route broadcast period of 120 seconds, link quality threshold of 70% for MintRoute, an awake period of 12 seconds, and a cycle period of 120 seconds (a duty cycle of 10%) for S-MAC. 81 Kbps was used as a transmission rate for this simulation. For the physical models, we used the basic standard configuration already provided by MiXiM defining AnalogueModels and Decider modules in a config.xml (Wessel et al., 2009). This included the necessary parameters such as the path loss, SINR-threshold and busy-threshold.

5.5 Simulation Results

In this section, we demonstrate the simulation results. Our experiments have been divided into two categories based on SensorScope deployments. At the beginning of the SensorScope project, a small number of nodes were deployed in order to evaluate the first use of multi-hop. Thus, our simulation environment consists of 7 nodes as described in Section 5.5.1. For the later SensorScope deployments, more nodes were deployed, therefore, making our simulation more complex. We increased the size of network topology to be 20 nodes as explained in Section 5.5.2. Furthermore, we further investigated the limitations of each simulation experiment by introducing a fault scenario into the network. As SensorScope discovered node failures during deployment, we injected dead nodes in the network simulation.

5.5.1 Experiment 1: introducing few nodes

Our first simulation experiment consisted of 7 nodes as shown in Figure 5.4.

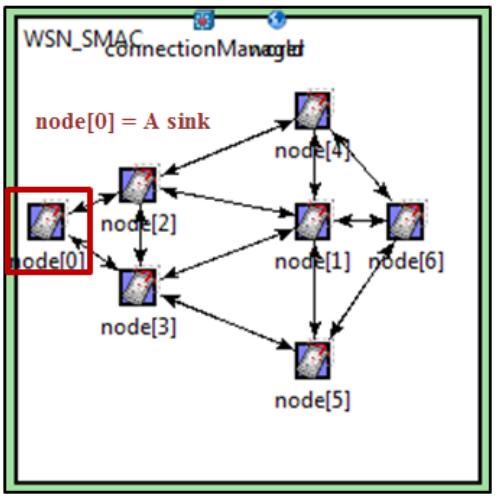


Figure 5.4: WSN topology for our case study

During the development and testing, simulation enables us to discover the failure in our developed communication protocols. Multiple running simulation experiments with random sample order were set to test the correctness of protocol models. For example, we indicated random distribution to the initial time of each node. This led to the different order of nodes performing its own operations during simulation at the slightly different starting time were assigned to each node. This resulted in variable traffic and packet ordering in the network that affected the different link quality and sequent route tree pattern.

From the affect of the random distribution we set into our running simulation experiments as described above, we discovered that 3 of 5 running experiments revealed the loop problem in the route tree of MintRoute protocol. To eliminate this problem, we redesigned and reimplemented our simulation model by introducing the loop detection mechanism suggested by the standard implementation of MintRoute (Woo et al., 2003). Figure 5.5 shows a snapshot of the result of the corrected simulation after running for 15000 seconds of total simulation time, excluding 710 seconds for the route setup. Different route trees were created based on the link quality towards the sink. An erroneous route tree was fully formed at time 710 seconds with a loop between node 4 and node 6 (as shown in Figure 5.5(a)). However, this loop problem is detected after node 4 receives its own packet back and then the new parent is selected. At time 1070 seconds, the quality of the link between node 4 and node 6 has dropped down below the threshold due to the packet loss caused by packet collision as we expect, hence, node 6 changes its parent from node 4 to node 5.

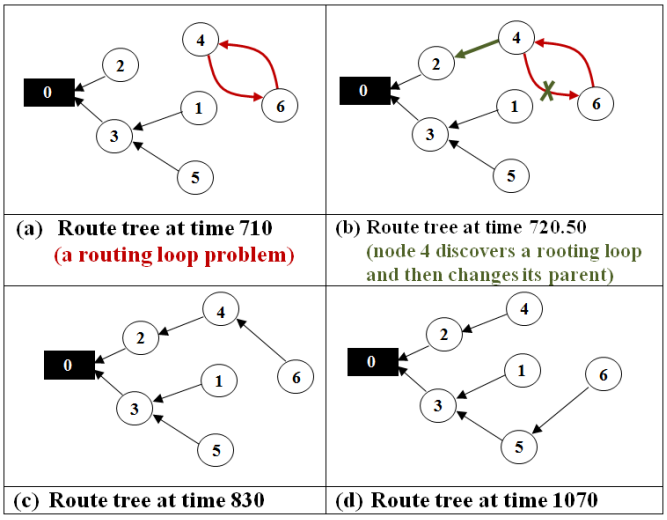


Figure 5.5: The different route trees created based on link quality

Figure 5.6(a) shows a number of packets received from sensor nodes at the sink. The number above the bar illustrated the average hop count for the whole simulation running time. Figure 5.6(b) shows the total quantity of sent data packets for each node, including all forwarded data packets due to the multi-hopping mechanism. Node 3 had the highest number of sent packets as it had many descendent nodes as shown in Figure 5.6(c). This led us to conduct another running experiment by introducing node 3 to be a dead node in order to replicate node failure as occurred in SensorScope project.

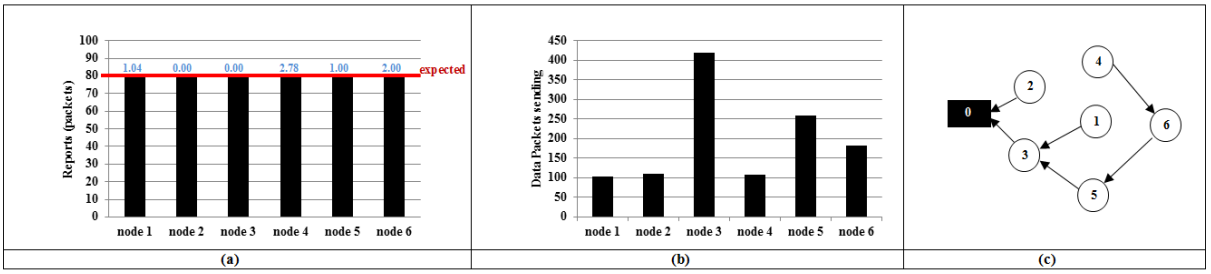


Figure 5.6: Result of the simulation: (a) data gathering reliability, (b) load distribution of the network and (c) route tree at time 11000 seconds

As node 3 had many descendent nodes, this led it to send a considerable number of packets. The latency was accomplished when the transmitted packet was received at the sink by measuring the time from source transmitting a packet to a sink receiving it. Thus, we investigated the data delivery delay by measuring the latency of each node. Figure 5.7 shows the latency of node 3 and some of its child nodes (5, 1) and grandchild node (4) measured at a sink, from the route tree at time 11000 seconds. From this evaluation, we also discovered that the latency was caused by a queuing delay. As a packet has to be queued during sleep period of S-MAC protocol, the duty cycle can affect the packet queuing delay.

Considering the measured latency shown in Figure 5.7, the first 35 packets of all nodes up to time 4200s were transmitted with low latency. This is because there was more balanced load distribution of transmitted packets over that simulation period. However, the route tree changed, leading the load distribution of transmitted packets to be less balanced at the later simulation period as shown in Figure 5.6(c). As node 5 had a better chance to transmit a packet before its sibling nodes (it indicated initial time earlier than others), the packets transmitted and forwarded from node 5 had little latency. This led to the bottleneck problem at node 1 as its latency increased considerably as shown in Figure 5.7(b).

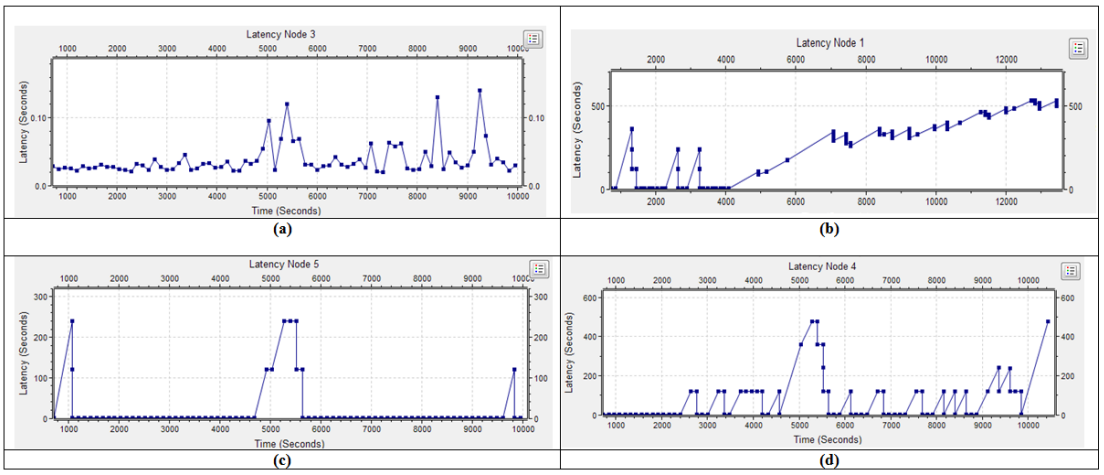


Figure 5.7: Latency of (a) node 3 and its child and grandchild nodes, (b-d) node 1, 5, and 4

Fault Scenario: the limitation of simulation was further investigated by introducing a fault scenario into another conducted simulation experiment. As can be seen from the result in the previous section, node 3 had the highest number of transmitted packets. This may lead node 3 to consume a lot of energy while transmitting these packets. To replicate dead nodes found in SensorScope project, we assumed that node 3 will have died eventually. Hence, node 3 was introduced as a failed node in our simulation model as illustrated in Figure 5.8(a). In our example, node 3 disconnects from the network after half the number of packets have been transmitted (at time 5500 seconds). We also found that a new route tree was created after node 3 died as demonstrated in Figure 5.8(a). However, half the number of node 3's packets were lost as shown in Figure 5.8(b). We also discovered that the packets from nodes 1 and 5 (2 and 3 packets of total 80 transmitted data packets at each node respectively) in transit in node 3's buffer after failure were lost because node 1 and 5 used node 3 as a forwarder to a sink before node 3 died.

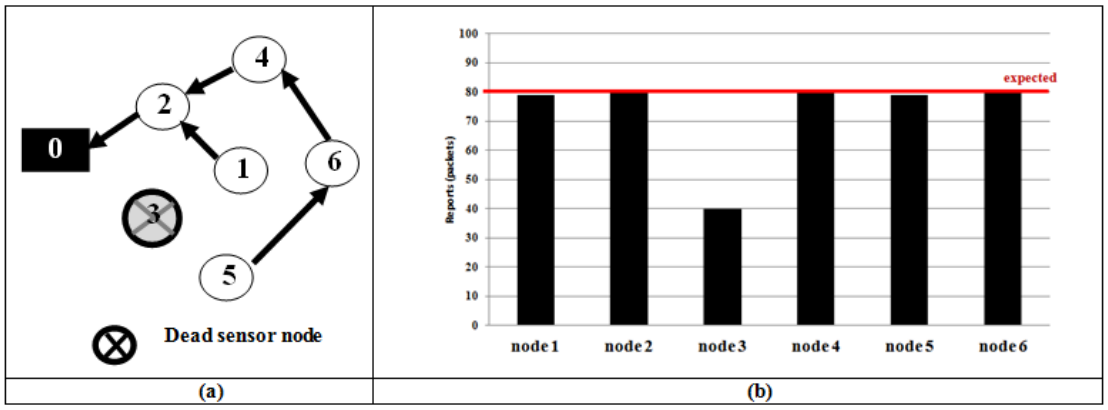


Figure 5.8: Example of fault scenarios: (a) route tree after node 3 died at time 5500 seconds (b) data gathering after node 3 died

5.5.2 Experiment 2: introducing more nodes with random location

We performed a further experiment which was expanded from the previous experiment. A network of 20 nodes was created with the location of each node randomly generated as shown in Figure 5.9.

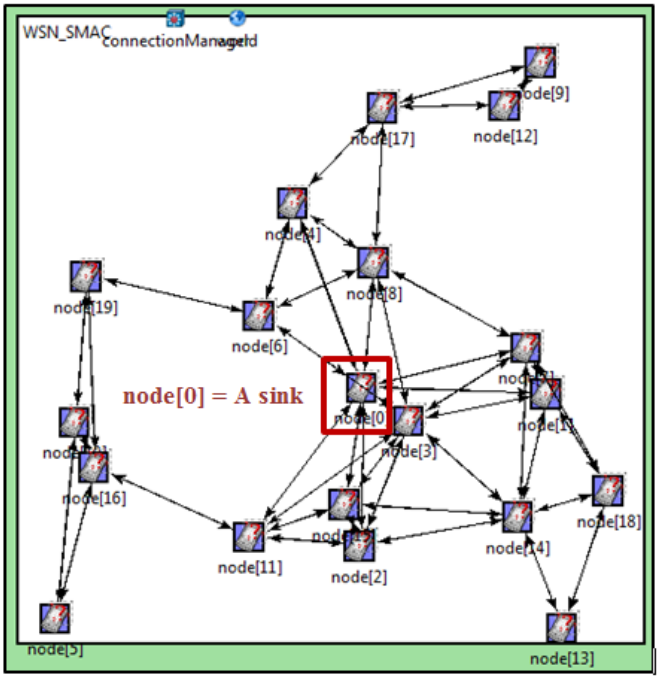


Figure 5.9: 20-node network topology for our case study

A route tree at time 5500s is shown in Figure 5.10(a). As can be seen in Figure 5.10(b), nodes 1, 6 and 11 were used as main nodes for transmitting packets to a sink. Thus, they had the high load distribution.

Fault Scenario: similar to the first experiment, fault injection was performed in another conducted simulation experiment. We assumed that nodes 1, 6 and 11 ran out of

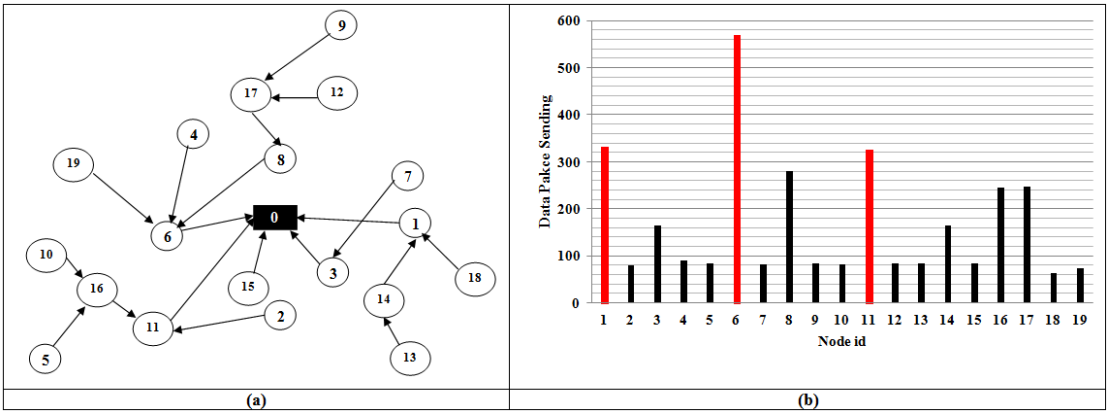


Figure 5.10: Result of the simulation: (a) route tree at time 5500 seconds, (b) load distribution at time 11000 seconds

battery eventually due to the high load distribution. Thus, we injected nodes 1, 6 and 11 to be disconnected from the network at time 5500 seconds. We also discovered that the route tree failed to be maintained as shown in Figure 5.11. Node 16 had no parent because it could not choose nodes 5, 19 and 10 and no other nodes in range to be its parent due to the loop problem. This caused our network graph to be disconnected.

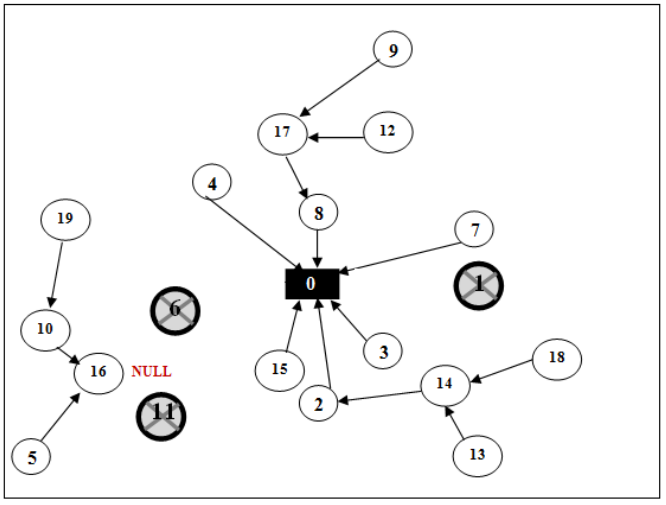


Figure 5.11: route tree after nodes 1, 6 and 11 died at time 5500 seconds

Although the main node for transmitting packets like nodes 1, 6 and 11 affected the route tree to be degraded, we discovered that the performance of the network was slightly decreased. 21.01% of total transmitted data packets were lost. Most lost packets were from nodes 5, 10, 19 and 16 because they required transit from dead nodes 6 and 11. Based on the result of the disconnected route tree, we found the interesting question to be analysed in which the different disconnected route tree may affect the number of packet loss. This led us to investigate more fault scenarios in order to ensure that the performance of the network still satisfies the level of packet loss. Figure 5.12 shows a route tree resulted in another running experiment that is different from the route tree

achieved from the previous experiment (shown in Figure 5.10(a)). This is because the different random distribution of initial time of each node affects the different route tree. As can be seen in this figure, nodes 3, 6 and 11 are main nodes for transmitting packets to a sink via this route. Thus, we injected these nodes to be dead nodes at the same time (5550 seconds) as shown in Figure 5.12(b). The result in Figure 5.12(b) shows that the route tree after some node died is better than that in the previous running experiment. In this example, the performance of the network was slightly reduced showing 18.71% of total transmitted packets were lost by a sink.

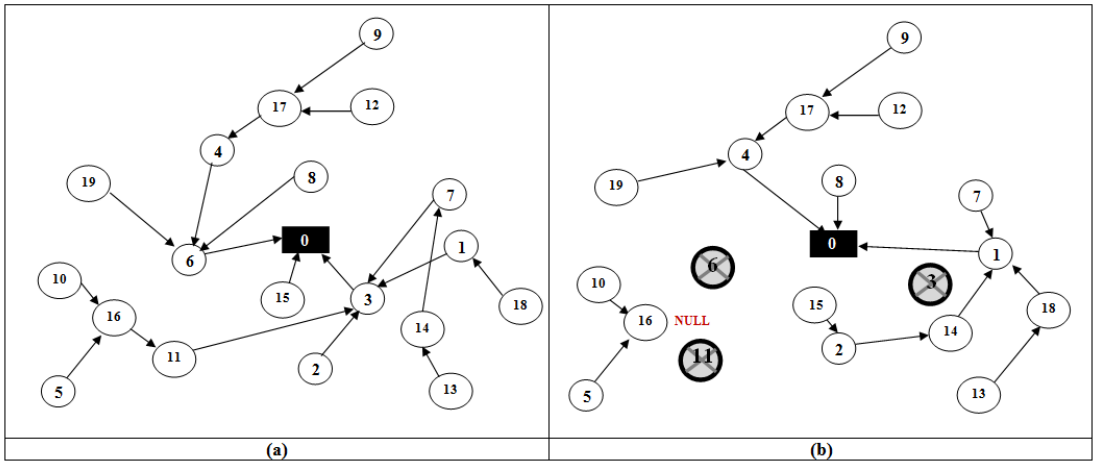


Figure 5.12: (a) route tree that has nodes 3, 11, 6 as main nodes for transmitting a forwarded packet to a sink, (b) route tree after nodes 3, 6 and 11 died at 5500 seconds

5.6 Conclusion

This chapter aims to explore the strengths and weaknesses of S-style development by implementing the case study proposed in the previous chapter. We have discovered some strengths of S-style development as mentioned in our research vision, Section 1.2. One of the main advantages of using simulation is that the high level of abstraction is enabled in which each protocol algorithm in the protocol stack is developed abstractly from the specific operating system and hardware platform. The performance of developed communication protocols and algorithm such as the network latency and the number of packet loss is analysed and evaluated by simulation described in Section 4.7 can be evaluated before the target code for the specific platform is generated. Furthermore, simulation can be used to reveal the load distribution of each node. The previous result enables us to reveal the traffic bottleneck problem causing traffic congestion.

A further advantage of using simulation is that simulation provides the concrete physical model and environment based on realistic assumptions. This influences the benefit in which the correctness of the network algorithms to be tested with the environment

scenario and condition generated by such an environment. Furthermore, this chapter has demonstrated the accurate modelling of stochastic lossy communication provided by the channel, radio and analogue model integrated in the physical layer in MiXiM. The interference wireless communication, received signal strength, signal-to-interference-plus-noise ratio (SINR) measurement provided by these physical models enables random lossy communication caused by the packet collision to be generated. This results in the link quality calculation of MintRoute protocol to be validated under realistic assumptions during the simulation.

In addition, we have demonstrated the successful modelling of failure and rerouting network graph disconnection scenarios generated by simulation. To replicate the failure in the SensorScope project, the node failure was modelled and injected into our simulation experiments in order to create disconnected network. The results of the failure experiments enable us to analyse whether the route tree can be recovered after the node failure. In case of the failure of a network causing the network to be split, we can analyse the degree of network partition by determining the level of packet loss as explained in Section 5.5.2. Finally, simulation also enables us to discover the loop problem occurring in the route tree.

Despite the strengths of simulation described above, we have discovered some weaknesses. Although simulation can reveal the fault in the protocol algorithm such as the loop problem in the route tree, this problem cannot be revealed in all running experiments we performed. Unlike simulation, formal methods can prove the absence of certain class of faults by proving the associated safety property is satisfied. This leads us to fix and prove this problem in formal models as explained in the next chapter. Thus, simulation cannot guarantee that the fault in the algorithm will be discovered.

Furthermore, this modelling style has no notion of refinement which causes complexity in development. Each single specific protocol algorithm e.g. SensorApp, MintRoute and S-MAC implemented at each layer in the protocol stack has to be completed together with environment elements provided by the standard interface in the simulation toolkit (wireless channel communication and connectivity, the library functions - implementing sending/receiving packet and packet encapsulation/decapsulation) to form a single simulation model. This leads us to encounter difficulties of managing such a complex simulation model (especially in MintRoute algorithm implemented at the network layer). The next chapter will demonstrate how to reduce this complexity by using Event-B modelling techniques. Each single controller for each protocol stack in a simulation model is separated and implemented into multiple layers by using the refinement technique.

Chapter 6

Formal Development (F-style Modelling)

6.1 Introduction

This chapter shows how formal methods can strengthen the simulation development methods that we introduce in Section 1.2, Chapter 1. We use Event-B, a proof-based formal language, to model the communication protocol algorithms. This demonstrates how the complexity in development can be reduced by using Event-B refinement and decomposition techniques. Furthermore, the correctness of protocol algorithms and especially the absence of certain classes of faults such as the loop problem can be proved.

Section 6.2 provides an overview of *SensorApp* and *MintRoute* requirements described in Section 4.3 before demonstrating Event-B development corresponding to these requirements in Section 6.3. The algorithms for both upper layer protocols are implemented through multiple layers of the refinement. To tackle the limitation of simulation discovered in the previous chapter, our developed formal model is encoded with the safety properties for proving the absence of certain classes of faults including the loop free properties. Furthermore, we implement a fault model by extending it from our Event-B model in order to introduce the unreliable link to validate MintRoute protocol. This fault implementation is described in Section 6.4. Section 6.5 discusses the Event-B development result and compares the benefits against limitations of Event-B development. In Section 6.6, we discuss the verification and validation techniques which we use while constructing the Event-B model before describing the result of ProB animation in Section 6.6.2.

Furthermore, the preparation step for co-simulation approaches is described in Section 6.7. This section demonstrates a shared-event decomposition technique for separating

the controller from the environment model. Then, Section 6.8 demonstrates the validation of achieved formal controller by using F-style co-simulation before deploying it in our C-style framework described in the next chapter. Finally, Section 6.9 discusses comparison of the strengths and weaknesses we discover among S-style simulation, model animation and F-style co-simulation testing techniques. Finally, Section 6.10 provides the conclusion.

6.2 An Overview of SensorApp and MintRoute Requirements

The focus of this chapter is the formal development and verification of two upper layers: SensorApp protocol for application layer and MintRoute protocol for network layer in Event-B. Here, the requirements that are used to implement our formal controller model are summarised in Table 6.1 and 6.2. The details of these requirements have been described in Section 4.3.

Table 6.1: A summary of functional requirements of SensorApp and MintRoute protocols

ID	Requirements
SensorApp protocol	
FUN-1	Sensor nodes' functionality:
FUN-1a	Each node senses data periodically and stores its sensed data in a sense buffer. Then, such data in the sense buffer is transmitted directly to its neighbour. After the sensed data is transmitted, it is deleted from sense buffer.
FUN-1b	the receiving sensor node records such packet into its receive buffer before forwarding this packet to the neighbouring node. Then, it clears the receive buffer after it transmits the packet to its neighbouring node.
FUN-2	A sink collects data from sensors. It has its own buffer for storing data. After the data is transmitted to a database server, it clears data from its buffer.
MintRoute protocol	
FUN-3a	Neighbourhood discovery
FUN-3b	The Link Quality Estimation
FUN-3c	Route Broadcast Mechanism
FUN-3d	Parent Selection
FUN-3e	Data Packet Forwarding Mechanism
FUN-3f	Failure Detection Mechanism

Table 6.2: A summary of non-functional requirements of SensorApp and MintRoute protocols

ID	Requirements
Network and link assumptions	
ENV-1	Nodes are connected by the dynamic links in which their quality may change at any time.
ENV-2	A data packet must be either transmitted successfully or lost from a source node to a sink in the dynamic network.
Reliability and fault tolerance requirements	
REL-1	The dynamic high-quality route requirements
REL-1a	MintRoute should find the best available high-quality route for data transfer
REL-1b	MintRoute should provide the disconnected/dead link detection mechanism so that the route tree must be recovered after the disconnected/dead link happens in the route tree
REL-2	S-MAC should provide the primitive implicit acknowledgement in order to improve the per-hop and end-to-end reliability of transmission.
Performance requirements	
PER-1a	The system must allow the clients to measure the load distribution of network.
PER-1b	The system must allow the clients to perform the network latency evaluation and analysis.
PER-2	the system must enable the clients to be able to measure and analyse the number of packet loss

Furthermore, the disconnected route tree discovered in the simulation model described in Chapter 5, showed that some, or all nodes may not be able to have a route constructed to the sink as shown in Figure 6.1. As a consequence, the additional system properties have been considered from requirements described above as follows.

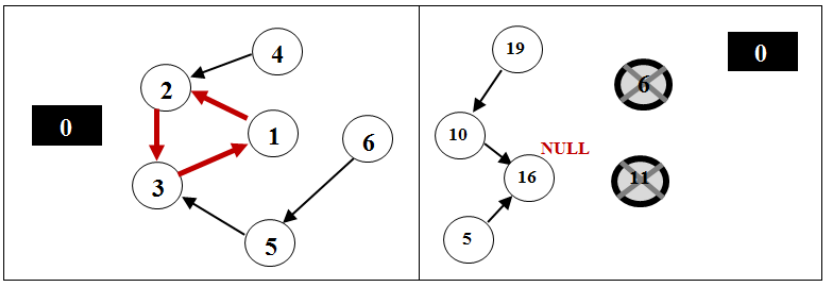


Figure 6.1: Problem: no node can reach the sink

We labelled these properties with **SAF** to indicate that they are safety properties of the system.

SAF-1: During the construction of the route tree, there are no loops in the route tree.

SAF-2: Once the route tree is completed:

SAF-2.1: each sensor node must have its own parent

SAF-2.2: each sensor node in the network must reach the sink.

6.3 WSN Development in Event-B

The benefit of using F-style for WSN development described in Section 1.2 is presented in this section. Event-B is used to create a WSN specification and its verification. The two upper layers: *Application* and *Network layers* are implemented. The formal model of these two protocols is given in Appendix A. The Event-B refinement technique is used to layer the model which corresponds to each layer of protocol stack. Six refinement models are created as shown in Figure 6.2. This begins with a very simple abstract model *M0* in which a data packet is transmitted to a sink in one atomic step. Then, the first and second refinement models (*M1-M2*) fulfil the operation for *SensorApp*. Model *M1* defines the neigbour node to determine multi-hop network for broadcasting mechanism before implementing network topology, sensing mechanism and channel mechanism in model *M2*. Then, this refinement model is refined down to implement *MintRoute* protocol. Each step of MintRoute protocol is layered (as in *M3-M5*) to implement four main mechanisms: *neighbourhood discovery* and *link quality estimation* (*M3*), *route broadcast mechanism* (*M4*) and *parent selection mechanism* (*M5*). Finally, we indicate the decomposition at two levels of interaction to enable a co-simulation. We apply shared event decomposition technique to separate the software controller model from the environment model at application and network layers. Thus, models *M2* and *M5* corresponding to these two layers respectively are decomposed as demonstrated in Figure 6.2. Model *M2* is decomposed into controller model *CTL2* and environment model *ENV2*, whereas model *M5* is split into controller model *CTL5* and environment model *ENV5*. The decomposition at these two layers is described in detail in Section 6.7.

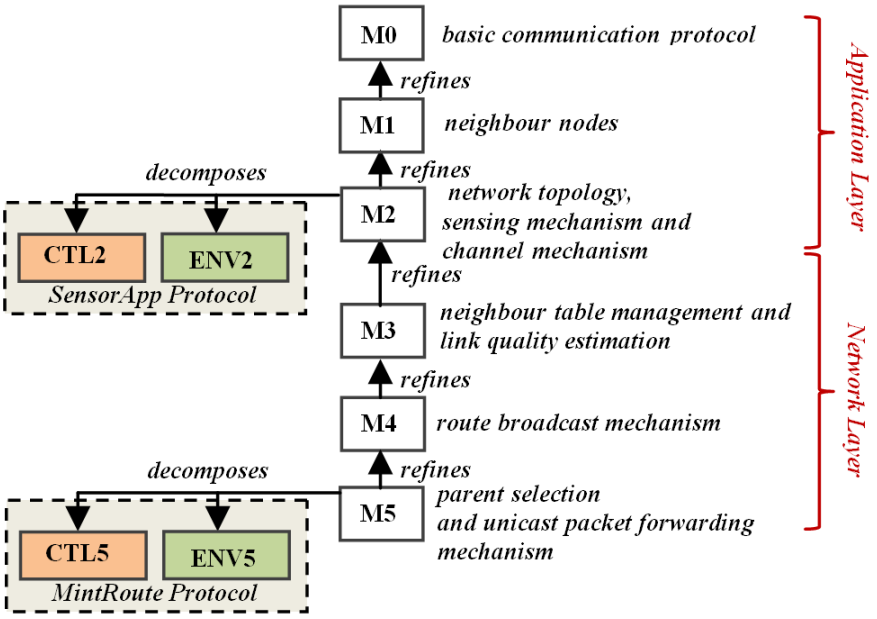


Figure 6.2: Event-B Development Approach

6.3.1 Initial Model (M0) - basic communication protocol

In this very abstract level, the basic communication protocol is implemented. This model enables the mechanism in which a data packet is transmitted from source nodes to a sink via the network abstractly as shown in Figure 6.3. The highlighted label and boxed label in this figure, represent corresponding events and variables respectively. In this abstract model, we consider **FUN-1** and **FUN-2** to implement the abstraction of data transmission from sensor nodes and the receiving data at a sink respectively.

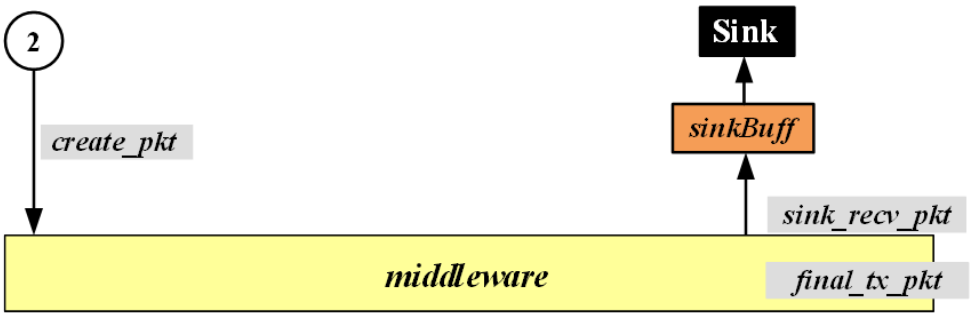


Figure 6.3: A flow of initial model behaviour

Static part of the initial model: we define *ND* as a finite set of nodes including a sink (*@axm0_1* - *@axm0_3*). Constant *PKT* is also defined as a set of packets. Axiom *@axm0_4* represents the initial source node of a packet. This constant is used in the receiving operation at the sink, implemented in the dynamic part, to check that the destination should not receive its own packet.

```
@axm0_1 ND ⊆ ℕ
@axm0_2 finite(ND)
@axm0_3 Sink ∈ ND
@axm0_4 initialSrcAddr ∈PKT → ND
```

Figure 6.4: The static properties of an abstract model

Dynamic part of the initial model: for the dynamic part of the initial model, variable *xmittedPkts* is introduced to represent transmitted packets in the network ($xmittedPkts \subseteq PKT$). Variables *sinkBuff* represents the receive buffer at the sink. Finally, the abstract variable *middleware* ($middleware \subseteq PKT$) represents packets in transit in the wireless medium.

Events: there are three events specified in this abstract model: *create_pkt*, *sink_recv_pkt* and *final_tx_pkt* for creating and transmitting a data packet from source node to a middleware, receiving a transmitted packet at sink and removing a packet from the middleware respectively.

6.3.2 Refinement Strategy

Before starting to explain the model of the system, it is beneficial to describe clearly the refinement strategy. We have divided the refinement strategy into three steps:

- **First refinement (M1):** introduces neighbour nodes into a network. In this refinement, we refine our abstract model to meet requirement **FUN-1b** by defining each sensor node to be a data router which forwards or broadcasts a packet to its neighbours.
- **Second refinement (M2):** takes network topology, sensing environment and channel mechanism into account. In this refinement, requirement **ENV-1** is considered to implement the network topology forming the links between nodes. As the sensing environment modelled in this refinement provides the environmental data for each sensing node, **FUN-1a** concerning the sense buffer operation is modelled.
- **Third refinement (M3):** concentrates on neighbourhood discovery and link quality estimation mechanisms for MintRoute protocol. This expands requirement **ENV-1** to implement more detail in the link quality (requirements **FUN-3a** and **FUN-3b**). Thus, a neighbour table and the link quality estimation for MintRoute protocol are introduced in this refinement.
- **Fourth refinement (M4):** extends the previous refinement model by adding the route broadcast (requirement **FUN-3c**) and dead neighbour detection mechanism (requirement **FUN-3f**) into this refinement model.
- **Fifth refinement (M5):** completes the MintRoute specification by implementing the parent selection (requirement **FUN-3d**) and the packet forwarding mechanisms via route (requirement **FUN-3**e). Thus, this refinement takes system properties **SAF-1** and **SAF-2** (including **SAF-2.1** and **SAF-2.2** into account).

6.3.3 First Refinement Model (M1) - neighbour nodes

The first refinement builds a more precise model than the initial model. The neighbour node is implemented to determine multi-hop network in this refinement. This means that the data packet can be forwarded hop by hop from source node to a sink. In this refinement model, three abstract events *create_pkt*, *sink_recv_pkt* and *final_tx_pkt* are refined in the refinement model and new events are added which enable the same observation as illustrated in the abstract model.

In Figure 6.5, the top of this figure demonstrates the observation of the abstraction. This starts with event *init* and then followed by events *create_pkt*, *sink_recv_pkt* and

final_tx_pkt respectively. On the bottom of the figure, the observation of the refinement model is expressed, which we divide into two sub-traces, (1) creating and transmitting packet and (2) forwarding a packet. For the creating and transmitting packet operation, event *create_dataPkt* refines event *create_pkt* to illustrate that a source node creates a data packet. Then, such a data packet is transmitted to its neighbours (event *start_tx*). As this refinement level introduces neighbour nodes, event *indicate_neighbours* indicates the neighbour nodes of the sender, as to whether receive a transmitted packet (*receive_pkt*) or discard a duplicated packet (*receive_dup_pkt*). This transmission process (accomplished by events *start_tx*, *indicate_neighbours*, *receive_pkt*/*receive_dup_pkt*) is repeated for forwarding a received packet if the receiving node is a non-destination node. The sink will receive a transmitted packet eventually during this forwarding process. Finally, to indicate the completion of packet transmission, the transmitted packet is removed from the middleware after the receiving node receives such a packet. As this refinement implements the packet forwarding mechanism in which a packet is transmitted from a source to a destination through hop-by-hop routing, two events are implemented in this refinement in order to accomplish the completion of each iterative transmission. The first is event *finish_tx_pkt* that indicates the finish of each round of iterative packet forwarding transmission (excepting the final one). This results in the current transmitted packet being removed from the wireless medium. The second is refining event *final_tx_pkt* from the previous abstract model. This event occurs at the final iteration of packet forwarding transmission in which the last transmitted packet is removed from the wireless medium.

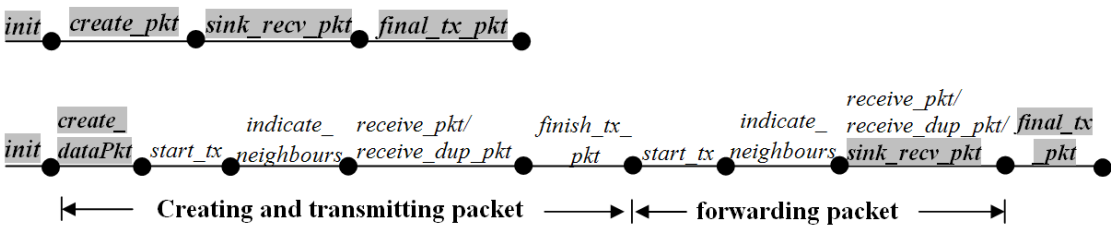


Figure 6.5: Initial abstraction and first refinement observations

Dynamic part of the first refinement model: In order to illustrate the relaying of transmitted packets from forwarder nodes to their neighbours, variable *ndBuff* is specified as the waiting buffer of each node (*@inv1_1*, Figure 6.6). We define variable *WiMedium* for the wireless transmission medium (*@inv1_2*, Figure 6.6). To create a more realistic application protocol, we replace variable *middleware* in the abstract model with variables *ndBuff* and *WiMedium* (*@glu1_1*, Figure 6.6). As the packet is in either the waiting buffer or the wireless medium, variables *ndBuff* and *WiMedium* must be disjoint (*@saf1_1*, Figure 6.6). In WSN, the neighbour list of a forwarder is specified by the physical environment to indicate the receiving packet mechanism. Variable *ctlNeighbours* is defined to recorded this neighbour list.

```
@inv1_1 ndBuff ∈ ND ↔PKT
@inv1_2 WiMedium ∈ ND ↔PKT
@inv1_3 ctlNeighbours ∈ PKT ↔ ND
@saf1_1 ndBuff ∩ WiMedium = ∅
@glu1_1 ran(ndBuff ∪ WiMedium) =middleware
```

Figure 6.6: Initial abstraction and first refinement observations

To gain more understanding, an example is used to demonstrate the operation of this refinement model as shown in Figures 6.7 and 6.8. Figure 6.7 shows a snapshot illustrating node 1 has data being transmitted to a sink via the wireless medium. In this case, node 1 creates a data packet and records it in a waiting buffer (*e1- create_dataPkt*) before such a packet is transmitted from waiting buffer to wireless medium (*e2- start_tx*).This event enables the transmitted packet to be removed from variable *ndBuff* and added into variable *WiMedium* as demonstrated in Figure 6.9(a). Since the routing mechanism has not been taken into account in this refinement, the data packet is broadcast from node 1 to all its neighbours (nodes 2, 3, 4, 5 and 6). To specify the neighbour node to receive a forwarded packet, the neighbour list of a forwarder is indicated non-deterministically (*e3*, *indicate_neighbours*), since the network topology has not been considered in this refinement. Figure 6.9(b) expresses event *indicate_neighbours* that is responsible for non-deterministic neighbour list assignment. This neighbour list is recorded in variable *ctlNeighbours*. The neighbour nodes will receive a transmitted packet (*e4*, *receive_pkt*). This event for receiving packet at non-sink nodes is expressed in Figure 6.9(c). As all receiving nodes are not a target node (*sink*), it records the received packet in the waiting buffer in order to forward it to a destination. Finally, since this transmission is not the final iteration of packet forwarding mechanism, event *finish_tx_pkt* (*e5*) is enabled to indicate the completion of this transmission by removing the order pairs between the current forwarder and that specific forwarded packet from the wireless medium.

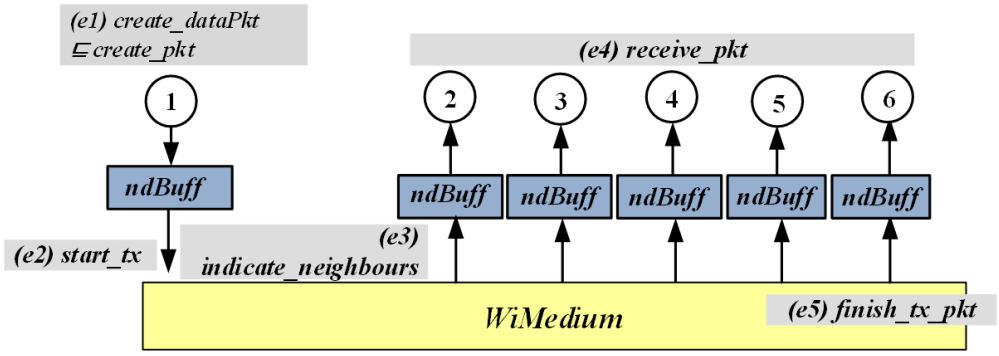


Figure 6.7: A flow of first refinement model behaviour for creating and transmitting packet mechanism

Figure 6.8 demonstrates that node 2 (one of the receiving neighbour of node 1) broadcasts the received packet in the waiting buffer to all its neighbours (1, 3, 4 and Sink). This enables the packet transmission process via channel to be repeated (*e2-e3*). In this

case, nodes 1, 3 and 4 detect the incoming data packet as a duplicated packet (*e5*, *receive_dup_pkt*) as node 1 receives its own packet and nodes 3 and 4 received this packet already. For the duplicated packet detection mechanism, the historical received packet of each node is recorded (in variable *floodTbl*) in order to detect the duplicated packet as shown in Figure 6.9(c). This results in the duplicated packet being discarded as demonstrated in Figure 6.9(d). Finally, the sink, as it is one of node 2's neighbours, receives this broadcast packet(*e6*, *sink_recv_pkt*). In this case, we assume that it is the final iteration of transmission. Thus, event *final_tx_pkt* (*e7*) is enabled to indicate the completion of this final transmission by removing the last transmitted packet from the wireless medium.

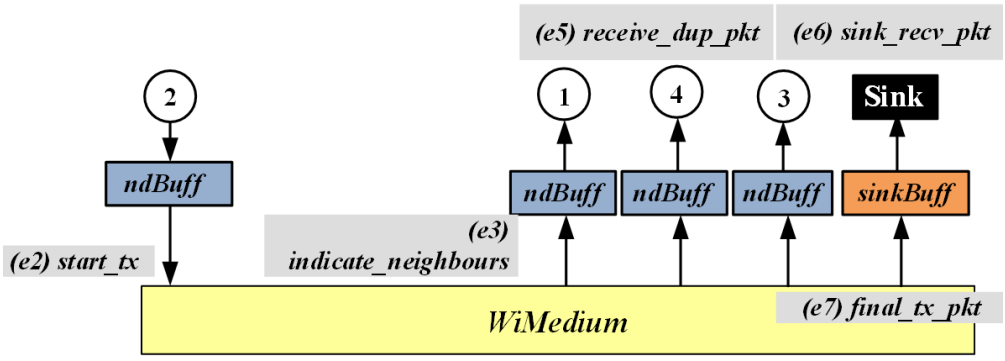


Figure 6.8: A flow of first refinement model behaviour for forwarding packet mechanism

```
event start_tx                                   (a)
any x pkt
  where
    @g1 x ↦ pkt ∈ ndBuffer
    @g2 x ↦ pkt ∉ WiMedium
  then
    @a1 ndBuffer := ndBuffer \{x↦pkt}
    @a2 WiMedium := WiMedium ∪ {x ↦ pkt}
 end
```

```
event indicate_neighbours                        (b)
 any pkt nbrs f
 where
   @g1 pkt ∈ ran(WiMedium)
   @g2 f ↦ pkt ∈ WiMedium
   @g3 nbrs ∈ {n|n ∈ (ℙ(ND)) }
   @g4 nbrs ≠ ∅
   @g5 pkt ∉ dom(ctlNeighbours)
 then
   @a2 ctlNeighbours := ctlNeighbours ∪({pkt}× nbrs)
end
```

```
event receive_pkt                                (c)
 any s nb pkt
 where
   @g0 pkt ∈ ran(WiMedium)
   @g1 pkt ↦ nb ∈ ctlNeighbours
   @g2 nb ↦ pkt ∉ ndBuffer
   @g3 nb ↦ pkt ∉ WiMedium
   @g4 s = initialSrcAddr(pkt) ∧ s ≠ nb
   @g5 nb ≠ Sink
   @g6 nb ∈ dom(floodTbl) ∧pkt ∉ floodTbl(nb)
then
   @a1 ndBuffer := ndBuffer ∪ {nb ↦pkt}
   @a2 ctlNeighbours := ctlNeighbours \ {pkt ↦nb}
   @a3 floodTbl(nb) := floodTbl(nb) ∪ {pkt}
 end
```

```
event receive_dup_pkt                            (d)
 any nb pkt
 where
   @g1 pkt ∈ dom(ctlNeighbours)
   @g2 pkt ↦ nb ∈ ctlNeighbours
   @g3 nb ∈ dom(floodTbl) ∧pkt ∈ floodTbl(nb)
 then
   @a1 ctlNeighbours := ctlNeighbours \ {pkt ↦nb}
 end
```

Figure 6.9: Some events defined in first refinement model that corresponds to a flow of first refinement model behaviour (a) *start_tx*, (b) *indicate_neighbours*, (c) *receive_pkt* and (d) *receive_dup_pkt*

6.3.4 Second Refinement Model (M2) - network topology, sensing mechanism, physical environment and channel mechanism

In this refinement, we elaborate our previous abstract model by introducing the physical environment model to complete SensorApp at application layer. This results in network topology, physical environment and channel mechanism being taken into account. Figure 6.10 demonstrates the structure of this refinement model. We apply the refinement atomicity technique (Butler, 2009; Salehi Fathabadi, 2012) to refine an abstract event in the previous abstract model (*M1*) into sub-atomic events in this refinement model (*M2*). Two main mechanisms are considered: *sensing* and *channel mechanisms* as shown in Figure 6.10(a) and (b) respectively.

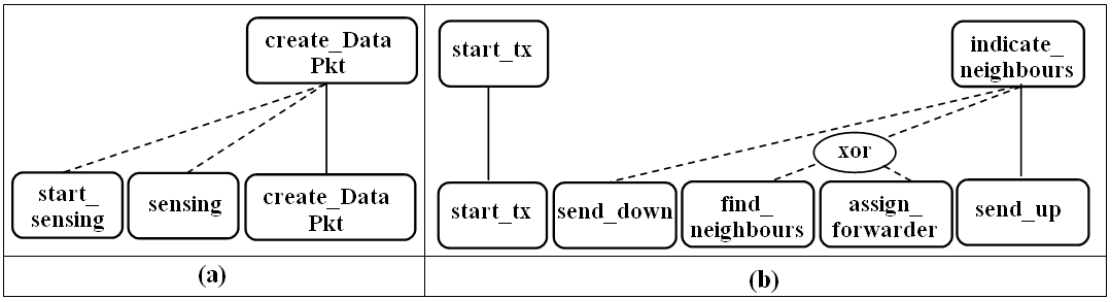


Figure 6.10: Some events of second refinement model

In Figure 6.11, the bottom of this figure shows the observation of this refinement model corresponding to that of its abstraction described in the previous refinement. Considering *sensing mechanism*, sub-atomic event *start_sensing* is enabled when there is data in the physical environment for each node to be sensed. Then, event *sensing* activates each node to sense data from the environment. This event results in the sensed data to be recorded in a sense buffer. When the node creates a data packet, data recorded in this sense buffer is attached to the data packet (accomplished by event *create_dataPkt*).

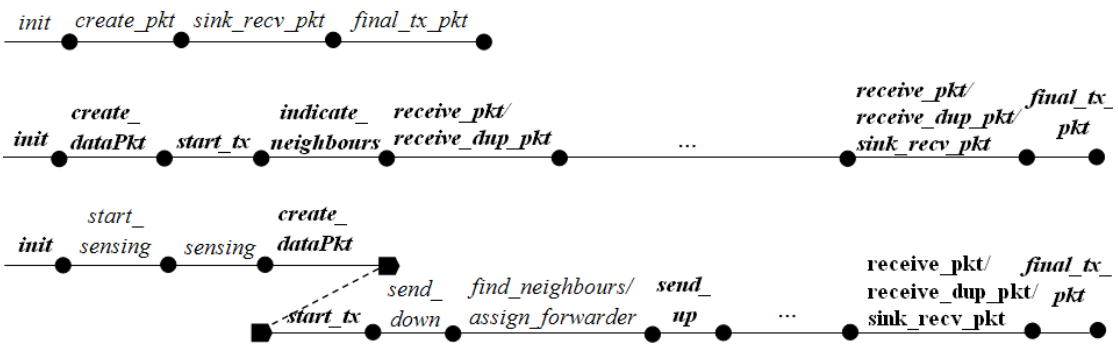


Figure 6.11: a trace of the second refinement behaviour

In the *channel mechanism*, a transmitted packet is relayed from source node down to lower layer before it arrives at a channel (by event *send_down*). As this refinement model implements the network topology, the specific neighbour of sender is specified from the

network topology. As a result, events *find_neighbours* and *assign_forwarder* is implemented to create a deterministic choice in which the former specifies either all neighbours or the latter identifies a specific neighbour of a sender in order to enable broadcast and unicast packet forwarding mechanism respectively. Then, the transmitted packet is relayed from a channel up to the lower layer of the neighbour (by event *send_up*). To implement this mechanism, we decomposed our abstract atomic event *indicate_neighbours* to sub-atomic concrete events *send_down*, *find_neighbours*, *assign_forwarder* and *send_up*. This results in a choice between two events: *find_neighbours* or *assign_forwarder* to identify the neighbour of sender.

Dynamic part of the second refinement model: we define variable *senseBuff* (*@inv2_1*, Figure 6.12) as a sense buffer for each node to store the sensed data. When the data packet is created, the sensed data in this variable is retrieved in order to be attached to such a created data packet.Variables *sentDown* and *sentUp* are defined for a set of packets transmitted down to adjacent lower layer protocol of sender and up to upper layer protocol of receiver respectively. In addition, variable *WiMedium* from the previous abstract model is replaced with these two variables as shown in gluing invariant *@glu2_1*. This invariant also demonstrates that variables *sentDown* and *sentUp* must be disjoint as the packet must not be sent up and down at the same time.

For physical environment development, variable *wsnLinks* is specified to represent the dynamic wireless sensor network topology. This variable contains the dynamic link between nodes (*@env2_1*) which has the property in which no node must not link directly to itself (*@env2_2*). As each node transmits a packet to each other through the wireless channel, variable *channel* is defined as shown in invariant *@env2_3*. We define variable *envNeighbours* (*@env2_4*) to record the deterministic neighbours identified from the network topology in the physical environment. This variable is prepared for the decomposition step in which the node controller model is separated from the physical environment model. This variable is used to synchronise the neighbour list specified by the physical environment with variable *ctlNeighbours* in the node controller model.

```
@inv2_1 senseBuff ∈ ND \{Sink} → ℙ(ℤ)
@inv2_2 sentUp ∈ ND ↔PKT
@inv2_3 sentDown ∈ ND ↔PKT
@glu2_1 partition(WiMedium, sentDown, sentUp)
@env2_1 wsnLinks ∈ WSN
@env2_2 (ND ◁ id) ∩ wsnLinks = ∅
@env2_3 channel ∈ ND ↔PKT
@env2_4 envNeighbours ∈ ran(channel) ↔ ND
```

Figure 6.12: Invariants of second refinement model

Figure 6.13 shows an example of the operations provided by this refinement model. *e1.1-e1.3* are sub-atomic events of event *create_dataPkt* (*e1*) to enable the sensing and creating data packet as described above. Then, the created packet is transmitted to the

lower layer of source node (*e2*). *e3.1*-*e.3.3* are the refinement atomicity of event defined in *e3* to specify the neighbour list more precisely. A transmitted packet is relayed from the lower layer to a channel (*e3.1*). Then, a channel looks up either all neighbours (*e3.2.1*) or a specific neighbour of the sender (*e3.2.2*) from a network topology. The result of this neighbour search mechanism is recorded in variable *envNeighbours* as demonstrated in Figure 6.14(a) and 6.14(b) for *find_neighbours* and *assign_forwarder* respectively. After the neighbours of the forwarder are identified, a relayed packet is forwarded to such neighbour via the upper layer (*e3.3*). As we decompose the node controller model from the physical environment model in the decomposition step, the neighbour list synchronization between these two models are required. In order to accomplish this, variable *ctlNeighbours* is needed to synchronise with variable *envNeighbours* in order to indicate the neighbour node controller to receive a transmitted packet. This neighbour list synchronisation is performed by event *send_up* as shown in Figure 6.14(c). Furthermore, this event also adds a relayed packet into variable *sentUp* and removes a corresponding packet from variables *sentDown* and *channel* as action *@a3*-*@a5* in Figure 6.14(c) respectively. Then, *e4*, the receiving mechanism is performed. Finally, as variable *WiMedium* is partitioned into variables *sentDown* and *sentUp*, either refining event *finish_tx_pkt* or *final_tx_pkt* (*e5*) is enabled to remove a transmitted packet in variable *sentUp* after each receiving node receives such a transmitted packet.

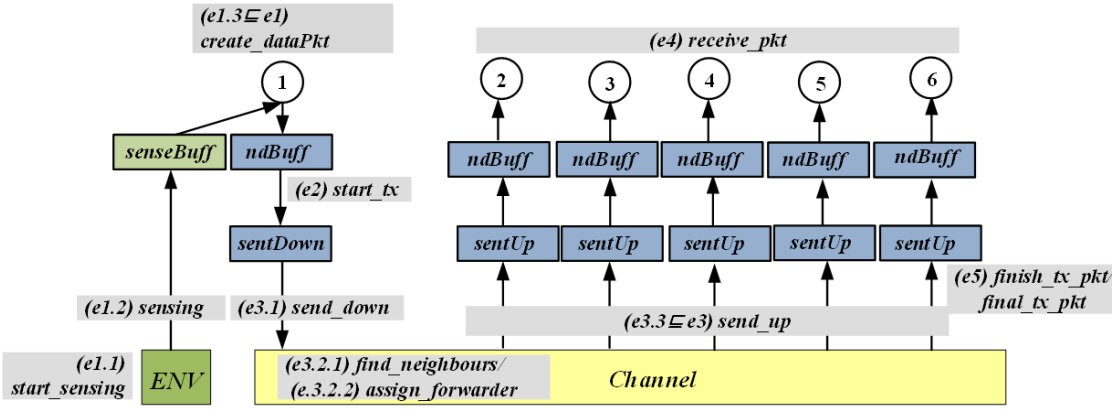


Figure 6.13: A flow of second refinement model behaviour for forwarding packet mechanism

```
event find_neighbour                                   (a)
any s nbs f pkt
  where
   @g1 channel ≠ ∅
   @g2 pkt ∈ ran(channel)
   @g3 s = initialSrcAddr(pkt)
   @g4 f ↦pkt ∈ channel
   @g5 nbs ⊆ ND ∧nbs = wsnLinks[{f}] ∧ nbs ≠ ∅
   @g6 envNeighbours = ∅
   ...
  then
   @a2 envNeighbours ≔ envNeighbours
            ∪ ({pkt} ×nbs)
   ...
 end
```

```
event assign_forwarder                                 (b)
any s nb pkt f nbs
  where
   @g1 channel ≠ ∅
   @g2 pkt ∈ ran(channel)
   @g3 s = initialSrcAddr(pkt)
   @g4 f ↦pkt ∈ channel
   @g5 nbs = wsnLinks[{f}] ∧ nbs ≠ ∅
   @g6 nb ∈ nbs
   ...
  then
   @a2 envNeighbours ≔ envNeighbours
            ∪ {pkt ↦ nb}
   ...
 end
```

```
event send_up                                          (c)
refines indicate_neighbours
  any pkt nbrs f
  where
   @g1 f ↦ pkt ∈ sentDown
   @g2 ran({pkt} ◁ envNeighbours) ∈ ℙ(ND)
   @g3 nbrs = ran({pkt} ◁ envNeighbours
   @g4 pkt ∉ dom(ctlNeighbours)
   @g5 f ↦ pkt ∉ sentUp
   @g6 f ↦ pkt ∈ channel
   ...
  then
   @a1 ctlNeighbours ≔ ctlNeighbours ∪({pkt}× nbrs)
   @a2 envNeighbours ≔ ∅
   @a3 channel ≔ channel \ {f ↦pkt}
   @a4 sentDown ≔ sentDown \ {f ↦ pkt}
   @a5 sentUp ≔ sentUp ∪ {f ↦ pkt}
   ...
 end
```

Figure 6.14: Events of second refinement model corresponding to neighbour list identification operation

6.3.5 Third Refinement Model (M3) - neighbour table management and link quality estimation

This refinement step refines the final model of SensorApp (M2) to develop the formal model for MintRoute protocol. We start with considering a neighbour table and link quality estimation of this routing protocol into account.

Dynamic part of the third refinement model: the neighbour table is defined as a relation between each node and its neighbours as shown in invariants *@inv3_1* in Figure 6.15. This neighbour table contains the number of lost packets (*missed*), the number of received packets (*received*), the sequence number of the last received beacon (*lastSeqno*) and the estimation of reception link (*receiveEst*), as shown in invariants *@inv3_2*-*@inv3_5* respectively.

As described in Section 4.5.1, an entry in a node's neighbour table is updated when the node receives a rebroadcast beacon packet from its neighbours. This beacon broadcast process starts when a sink broadcasts a beacon packet to its neighbours periodically. The receiving neighbours then rebroadcast a received beacon packet to their neighbours. This

```
@inv3_1 neighbourTbl ∈ ND ↔ ND
@inv3_2 missed ∈ neighbourTbl → ℕ
@inv3_3 received ∈ neighbourTbl → ℕ
@inv3_4 lastSeqno ∈ neighbourTbl →ℕ
@inv3_5 receiveEst ∈ neighbourTbl → ℕ
```

Figure 6.15: Invariant defining the neighbour table structure

rebroadcasting process is repeated until every sensor node in the network rebroadcasts a beacon packet. Figure 6.16 demonstrates an example of the beacon broadcast process illustrated by a sink. When nodes 2 and 3, the sink's neighbours, receive each broadcast beacon packet from a sink, they will add a sink into their neighbour table. Since it is the first beacon transmitted from a sink, the fields corresponding to a sink record are assigned as shown in this figure. Note that field *receiveEst* is recorded to be zero as the link quality has not been estimated in this step.

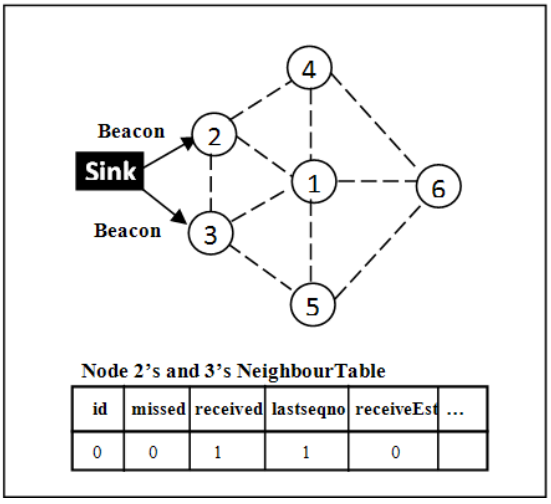


Figure 6.16: Neighbour table structure

A beacon broadcast mechanism: to implement a beacon broadcast mechanism, we refine event *create_packet* defined in the application model to implement *create_bconPkt*. This enables the beacon packet transmission mechanism to be able to use the same channel with the data packet transmission mechanism.

Neighbourhood discovery for neighbour table discovery: we create two main events *add_newEntry* and *update_nbr*.

- Event *add_newEntry* is enabled when a neighbour node does not exist in the neighbour table. In this case, it adds such a neighbour into the table and set all corresponding entries in the neighbour of that neighbour to be zero.
- Event *update_nbr* is responsible for updating the existing entry of the neighbour into the neighbour table when a broadcast packet is received. This event calculates

and updates three main values into the neighbour table: *received*, *missed* and *lastSeqno*.

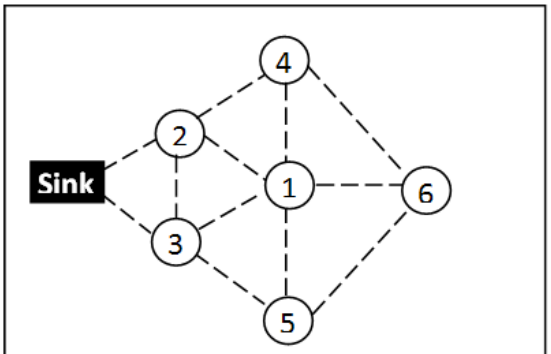
- A number of received beacons (*received*): this number is calculated when node x receives a beacon packet from its neighbour node y. This results in the number of received beacon from node y recorded in the neighbour table to be incremented. This is demonstrated as action $received(x \mapsto y) := received(x \mapsto y) + 1$.
- A number of lost beacons (*missed*): it is calculated as $missed_{x-y} = sNo - (lastSeqno(x \mapsto y) + 1)$. Parameter *sNo* indicates the sequence number attached to the current received beacon at node x from node y. Parameter $lastSeqno(x \mapsto y)$ represents the sequence number of the last received beacon from node y recorded in the neighbour table of node x. For example, in node x's neighbour table, the recorded sequence of the last received beacon from node y is 1 ($lastSeqno(1 \mapsto 2) = 1$). Then, node 1 receives the beacon which has the sequence number to be 4. In this case, $sNo = 4$. Thus, the number of lost packets between node 1 and 2 ($missed_{1-2}$) is calculated as 4-(1+1) =2
- The sequence number of the last received beacon (*lastSeqno*): after the number of received packets and the number of lost packets are calculated, the sequence number attached to the current received beacon is updated into field *lastSeqno* of the corresponding record in the neighbour table. In the example demonstrated above, the sequence number of current received beacon (*sNo*) at node x from node y (*sNo*), that is 4, is recorded as the sequence number of the last received beacon from node y in the neighbour table of node x. We implement this calculation as action $lastSeqno := lastSeqno \lhd\!\!\!- \{x \mapsto y \mapsto sNo\}$.

Link Quality Estimation: each node estimates periodically the reception ratio as an inbound link quality based on the successful rate of transmitted beacon packets as shown in the equation below.

$$PR_{xy}(t-1,t) = \frac{the\ number\ of\ received\ beacons\ by\ x\ from\ y\ in\ (t-1,t)}{the\ number\ of\ actual\ rebroadcast\ beacon\ to\ x\ from\ y\ in\ (t-1,t)} \times 100 \tag{6.1}$$

This equation illustrates that this reception ratio of node y is performed by node x every period *(t-1,t)*. In MintRoute, each node has to rebroadcast the same beacon initiated by a sink within the periodic beacon broadcast period. As a result, the number of rebroadcast beacons of each node in period *(t-1,t)* should be the same. Consequently, the node (e.g. node x in the above equation) can indicate the number of actual rebroadcast beacon from its neighbours (e.g. node y) during *(t-1,t)* by using this number.

As shown in Figure 6.17, for example, we assume that each node actually transmits five beacons during periodic *(t-1,t)* period. However, node 1 receives four and three beacons from nodes 2 and 3 respectively. Therefore, the reception link quality of nodes 2 and 3 is 80% and 60% respectively.



Node 1's NeighbourTable

id	missed	received	lastseqno	receivedEst	...
2	1	4	5	80	
3	2	3	5	60	
...					

Figure 6.17: Example of link quality estimation

In this refinement model, we implement this reception ratio calculation in event *update_est*. We encode the calculation as shown in (6.1) as a guard in this event. It is implemented as $(r\ /\ rt)\ *\ 100$, where r is the number of received beacons and rt is the number of actual rebroadcast beacons. This means that r is divided by rt first before the result of the division is multiplied by 100. However, as the result of a natural or an integer number division in Event-B is always rounded towards zero, this leads to the incorrect calculation e.g. $(2/5) * 100 = 0$ as same as $(3/5) * 100$ instead of 40 and 60 respectively. We correct this by introducing r to be multiplied by 100 first and then the result is divided by rt $((r\ *\ 100)\ /\ rt)$.

In order to round off the error caused by the integer division, we measure the relative error. This relative error is calculated as demonstrated in (6.2). Note that x/y denotes the real division and $\left[x/y\right]$ represents the integer division.

$$\frac{\left|r\ /\ rt\ -\ \left[r\ /\ rt\right]\right|}{|r\ /\ rt|} \tag{6.2}$$

Based on the equation above, 100% of error caused by the two number division in Event-B happens, when dividend r is less than divisor rt. For example, let $r = 1$ and $rf = 3$, we find that the relative error of this two number division is 1 or 100% ((0.33 - 0)/0.33). To reduce the chance of this error, dividend r needs to be multiplied by the power of

ten as 10^n, where $n \geq 1$. From the example described earlier, we find that the relative error is reduced significantly from 100% to approximately 10% ((3.33 - 3)/3.33) and 1% ((33.33 - 33)/33.33), when dividend r is multiplied with 10 and 100 respectively. (6.3) demonstrates an equation for calculating the relative error of the two number division rounded off with multiplier 100.

$$\frac{\left|(r * 100)/\, rt \; - \; \left[(r * 100)/\, rt\right]\right|}{|(r * 100\,)/\, rt\,|} \tag{6.3}$$

Therefore, to obtain more accurate division result, we indicate 100 to be multiplied with dividend r. Furthermore, by using this number, the result of division is converted to a percentage. This meets the link quality calculation requirements in the MintRoute protocol in which the link quality is possibly considered as either a ratio or a percentage.

6.3.6 Fourth Refinement Model (M4) - route broadcast mechanism

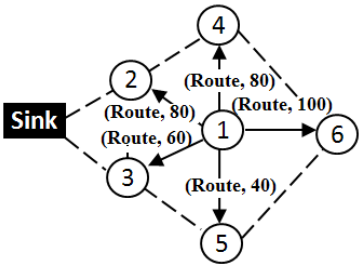
In this refinement, we consider the route broadcast mechanism. Figure 6.18 shows an example of route broadcast mechanism. After each node estimates the reception link quality (*receiveEst*) of each neighbour in the neighbour table, it broadcasts a route update packet to its neighbours. We assume that the neighbour table of node 1 containing the calculated *receiveEst* of each neighbour illustrated in Figure 6.18(a). It creates and broadcasts a route update packet to its neighbour as shown in Figure 6.18(b). For example, node 1 retrieves the receiving information of node 2 from its neighbour table in which the quality (*receiveEst*) is 80%. Then, it attaches this number to the route update packet and transmits it to node 2. When the node receives the route update message from the node identified in the neigbour table, it updates the corresponding entry with the value attached in the packet. Especially, the reception link quality (*receiveEst*) specified in the packet is recorded as the transmission link quality (*sentEst*). As can be seen in Figure 6.18(c), after node 2 receives the route update packet from node 1 with the attached *receiveEst* that is 80%. It updates its neighbour table with this *receiveEst* as the sending quality from itself to node 1 (*sentEst*).

Dynamic part of the fourth refinement model: there are two main variables introduced to this refinement. These variables are an extended variable from the neighbour table defined in the previous refinement model (as in Figure 6.15). Figure 6.19 demonstrates typing invariants defining these extended variables. One variable is *sentEst* (*@inv4_1*) which corresponds to the transmission ratio in the neighbour table. Another is *liveliness* (*@inv4_2*) which corresponds to the field *liveliness* in the neighbour table. Each node uses this variable as a counter of neighbours life time. This value is updated every time when the node receives a route update packet from its neighbours. However, when this variable becomes zero, it means that this neighbour may disconnect

(a) Node 1's NeighbourTable after estimating receiving quality (*receiveEst*) of its neighbours

id	missed	received	lastSeqno	receiveEst	...
2	1	4	5	80	
3	2	3	5	60	
4	1	4	5	80	
5	3	5	5	40	
6	0	5	5	100	

(b) node 1 broadcasts a route update packet to its neighbours



(R,r): R = Route update packet
r = reception ratio (*receiveEst*)

(c) Node 2's NeighbourTable after receiving a route broadcast packet from node 1

id	...	receiveEst	sendEst	...
1	...	100	80	
...				

Figure 6.18: Example of broadcasting route update packet process

from the network or die. This results in this neighbour to be excluded from any process of MintRoute algorithm.

```
@inv_4_1 sentEst ∈ neighbourTbl → ℕ
@inv_4_2 liveliness ∈ neighbourTbl → ℕ
```

Figure 6.19: Typing invariant defining the transmission ratio (*sentEst*) and field liveliness (*liveliness*) of the neighbour table

Events: to accomplish this refinement, we refine event *add_newEntry* and implement three new events: *update_route*, *decrease_liveliness* and *exclude_deadNbr*.

- Event *add_newEntry* sets *sentEst* to be zero and *liveliness* to be a maximum threshold (*LIVELINESS*) after the neighbour is added into the neighbour table.
- Event *update_route* is responsible for updating the existing entry of *sentEst* into the neighbour table. This event also resets variable liveliness back to *LIVELINESS* threshold.
- Event *decrease_liveliness* is enabled when the link quality estimation mechanism occurs (event *update_est* is enabled). This causes this variable to be reduced by 1 ($liveliness := liveliness \lhd\!- \{x \mapsto y \mapsto (live - 1)\}$, where $live = liveliness(x \mapsto y)$).
- Event *exclude_deadNbr* is responsible for detecting a dead neighbour in the neighbour table. The dead neighbour is indicated when its liveliness value becomes zero.

6.3.7 Fifth Refinement Model (M5) - parent selection and unicast packet forwarding mechanism

This refinement step implements the parent selection mechanism, performing periodically to specify one of node's neighbours as a parent for routing. The path cost towards a sink is calculated based on an inverse of the multiplication of both two ratios, *reception* and *transmission*, ($\frac{1}{receiveEst_{(x,j)}} \times \frac{1}{sentEst_{(x,j)}}$, *where x and j indicates node x and its neighbours j respectively*). The poor quality link that has both ratios less than the link quality threshold is not considered. A neighbour with the smallest path cost is chosen as a parent.

Figure 6.20 demonstrates an example of the parent selection process. The possible *receiveEst* and *sentEst* of the link between node 1 and its neighbour is calculated as shown in Figure 6.20(a). In the parent process, this link quality threshold is set to 80%. Thus, in this example, the links between this node to nodes 3 and 5 are not considered as demonstrated in Figure 6.20(b). The link between node 1 and 6 has the highest link quality. However, node 1 does not choose node 6 to be a parent because this node does not have the smallest path cost as it is furthest away from a sink. Node 2 that has the next highest link quality but the smallest path cost is chosen as a parent of node 1 as can be seen in Figure 6.20(c).

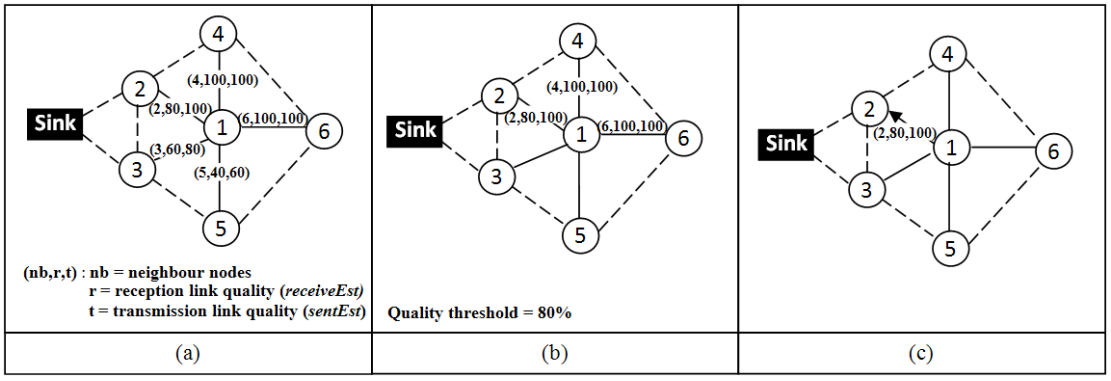


Figure 6.20: Example of Parent Selection process

Static part of the fifth refinement model: we applied the the definition of transitive closure proposed by Abrial (2010) and applied in Damchoom et al. (2008) in order to illustrate all reachable paths in the network (*@axm_tcl_1* - *@axm_tcl_4*) as shown in Figure 6.21. We define the transitive closure (*tcl*) as the total function between WSN and itself (*@axm_tcl_1*). Axiom *@axm_tcl_2* identifies the property in which for all relation *r* of WSN, *r* is included in *tcl(r)*. Axiom *@axm_tcl_3* is the property illustrating the forward composition of *tcl(r)* with *r* is included in *tcl(r)*. Finally, axiom *@axm_tcl_4* represents the property in which *tcl(r)* is the smallest relation. An example of transitive closure can be seen in Figure 6.22.

```
@axm_tcl_1 tcl ∈ WSN → WSN
@axm_tcl_2 ∀r·(r∈WSN ⇒  r ⊆ tcl(r))
@axm_tcl_3 ∀r·(r∈WSN ⇒  r;tcl(r) ⊆ tcl(r))
@axm_tcl_4 ∀r,t·(r∈WSN ∧ r⊆t ∧ r;t⊆t  ⇒ tcl(r)⊆t)
```

Figure 6.21: Transitive Closure (TCL) properties

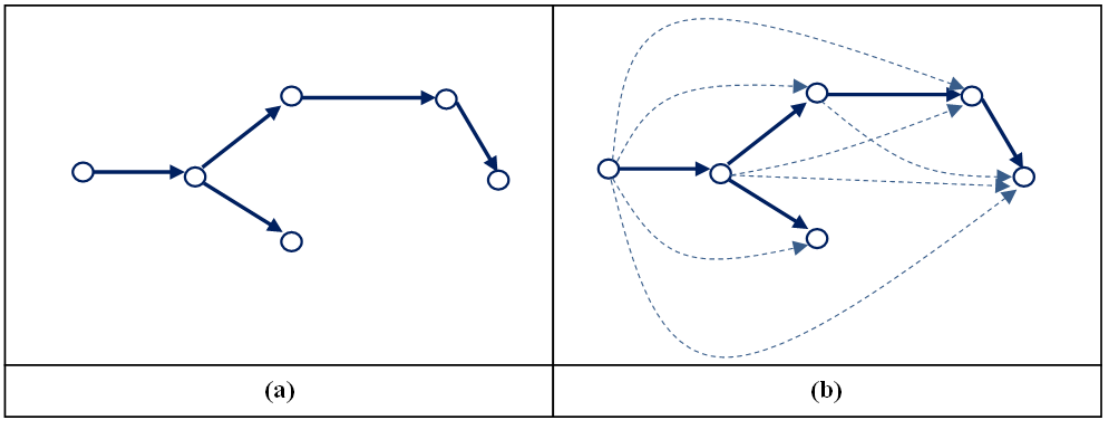


Figure 6.22: Transitive closure constructs the output graph (b) from the input graph (a)

Dynamic part of the fifth refinement model: first of all, we extend the neighbour table from the previous refinement model to implement fields parent of the neighbour (@*inv*5_1 in Figure 6.23) and its path cost towards a sink (@*inv*5_2 in Figure 6.23).

```
@inv5_1 parent ∈ neighbourTbl → ND ∪ {BROADCAST}
@inv5_2 cost ∈ neighbourTbl → ℕ
```

Figure 6.23: Typing invariant defining fields parent of the neighbour (*parent*) and its path cost towards a sink (*cost*) of the neighbour table

For the route tree construction verification, we create safety invariants as shown in Figure 6.24. This shows us the benefit of proof to indicate no-loop property as a necessary invariant proved after simulation revealed it. We define the route tree in invariant *@inv5_3*. The initialisation of variable nodes contains only a sink (*Sink*) and the route tree (*cRouteTree*) is initialised to be an empty set. As soon as a node is explored to discover its parent, it is recorded in nodes with a pair between this node and its parent recorded in variable *cRouteTree*. Safety invariant *@saf5_1* demonstrates the 'no loop' property. This property is applied from the no-loop property proposed by Abrial in Abrial (2010). It is introduced in order to ensure that there are no loops in a routing tree (satisfying **SAF-1** described in Section 6.2). We define S in this invariant as the set which consists of a cycle or loop in a route tree. $cRouteTree^{-1}[S]$ is defined to indicate the direct descendants of all element of set S. The loop in the route tree occurs when $S \subseteq cRouteTree^{-1}[S]$. Consequently, to prevent the loop in the route tree, S must be an empty set. This can then guarantee that there are no loops in the routing tree.

Theorem *@mth5_3* confirms every node must have a route to a sink. Theorems *@mth5_1* and *@mth5_2* are introduced to help the proof of *@mth5_3*.

```
@inv5_3 cRouteTree ∈ (nodes \ {Sink})→ ND
@saf5_1 ∀S · S ⊆ cRouteTree~[S] ⇒ S = ∅
theorem @mth5_1 ∀T·(Sink∈T ∧ cRouteTree~[T] ⊆ T ⇒ nodes ⊆ T)
theorem @mth5_2 nodes ⊆ {Sink} ∪ (tcl(cRouteTree))~[{Sink}]
theorem @mth5_3 nodes \ {Sink} ⊆ (tcl(cRouteTree))~[{Sink}]
```

Figure 6.24: Safety invariants regarding the route tree in the fifth refinement model (*M5*)

Figure 6.25 shows the main events corresponding to the parent selection process.

```
event choose_node                                   (a)
  any x nb
  where
   @g1 x ∈ ND \ {Sink}
   @g2 x ∉ chosenNDs
   @g3 nb = ran({x} ◁ neighbourTbl)
        ∧ nb ≠ ∅
   @g4 Nbrs = ∅
   ...
then
   @a1 chosenNDs := chosenNDs ∪ {x}
   @a2 Nbrs := nb
end
```

```
event cal_PCost_gt                                  (b)
any x nb pCost nbs rEst sEst lnkQty th
where
 @g1 x = chosenNDs ∧ nb ∈ Nbrs
 @g2 x ↦ nb ∈ neighbourTbl
 @g3 x ↦ nb ∈ dom(receiveEst) ∧ receiveEst(x ↦ nb) ≥ th
 @g4 x ↦ nb ∈ dom(sentEst) ∧ sentEst(x ↦ nb) ≥ th
 @g5 rEst = receiveEst(x ↦ nb) ∧ sEst = sentEst(x ↦ nb)
 @g6 lnkQty ∈ ℕ ∧ lnkQty = ((120 − rEst) ∗ (120 − sEst))
 @g9 nbs = ran({x} ◁ neighbourTbl) ∧ nbs ≠ ∅
 @g10 pCost ∈ ℕ ∧ pCost = cpCost (nb) ∧ pCost ≠ MAX_DIST
 @g11 x ↦ nb ∉ calPCostNDs
 @g12 pCost + lnkQty ≥ 0
then
 @a1 calPCostNDs := calPCostNDs ∪ {x ↦ nb}
 @a2 calPCost := calPCost ∪ {x ↦ nb ↦ (pCost + lnkQty)}
 @a3 Nbrs := Nbrs \ {nb}
end
```

```
event choose_parent                                 (c)
  any x p m des
  where
   @g1 p ∈ ND ∧ x ∈ nodes \ {Sink}
   @g2 x ∉ dom(cRouteTree)
   @g3 ran(calPCost) ≠ ∅
   @g4 m = min(ran(calPCost))
   @g5 p ∈ ran(dom(calPCost ▷ {m}))
   @g6 x ≠ p
   @g7 des ⊆ nodes
   @g8 des = (tcl(cRouteTree))~[{p}]
   @g9 p ∉ des ∪ {x}
   ...
  then
   @a1 cRouteTree := cRouteTree ∪ {x ↦ p}
   @a2 nodes := nodes ∪ {x}
   @a3 cpCost(x) := m
   ...
 end
```

Figure 6.25: Main events for performing parent selection mechanism in the fifth refinement model

- Event *choose_node* begins choosing node x that has not chosen its parent (guard *@g2*, Figure 6.25(a)). We define variable *chosenNDs* to keep a list of non-deterministically chosen nodes that has not identified a parent (as in action *@a1*, Figure 6.25(a)). This results in the selected node's neighbours to be identified (guard *@g3* and action *@a2*, Figure 6.25(a)) for path cost calculation.

- Event *cal_PCost_gt* integrates the calculation of the path cost towards a sink of each neighbour node. This event is divided into three steps: *choosing neighbour*

nodes, *calculating the link quality of each neighbour* (*linkQty*) and *calculating path cost* (*calPCost*).

- *Choosing neighbour nodes*: only a neighbour node that has good link quality is included in this process. This means that its *receiveEst* and *sentEst* must meet the minimum of threshold *th* as shown in guards *@g3-4*, Figure 6.25(b).
- *Calculating the link quality of each neighbour (lnkQty)*: the value of these two link quality ratios are used to determine the link quality of each neighbour as illustrated in guard *@g6*, Figure 6.25(b).
- *Calculating path cost (calPCost)*: this path cost is calculated from the sum of the numbers between the parent cost (*pCost*) and the link quality of each neighbour (*lnkQty*) retrieved from the neighbour table as expressed in guards *@g10-11*, Figure 6.25(b). Note that this calculation does not consider the neighbour node that is far away from a sink (has path cost as *MAX_DIST*). The result of calculation is recorded in variable *calPCost* (action *@a2*, Figure 6.25(b)).

- Event *choose_parent* identifies a parent of the selected node by considering its neighbour that has the smallest path cost is chosen as a parent. We introduce variable x as a node that considers one of its neighbours to be a parent p. The parent p of node x is identified as the neighbour node that has the smallest path cost (guard *@g4* and *@g5*, Figure 6.25(c)). This results in the order pair between x and p to be added into the route tree (variable *cRouteTree*) as shown in action *@a1*, Figure 6.25(c).

  To prevent the loop problem in the route tree, we define *des* as a set of all ancestor nodes along the route from node x towards a sink. The *tcl* function is used to identify these ancestor nodes (as shown in guard *@g8*). In order to guarantee that there are no loops in the route tree, the chosen parent p must not be node x itself or any ancestor nodes as shown in guard *@g9*. Since these guards prevent the loop to be added into the route tree, set S defined in invariant *@saf5_1* is always empty. This leads event *choose_parent* to always satisfy this invariant.

Packet forwarding mechanism via route: the constructed route tree is used to route a data packet from source node to a sink. The parent of sender/forwarder identified in this route tree is used as a next node to transmit a data packet. In refinement model *M1*, we define event *start_tx* to perform the packet transmission and forwarding mechanism from source/forwarder node to a sink as explained in Section 6.3.3. Thus, to implement this data packet forwarding mechanism, we refine this event to implement two events: *start_tx_dataPkt_bct* and *start_tx_dataPkt_fwd*. The former is responsible to broadcast a data packet, if the sender/forwarder has not discovered its parent. It assigns the broadcast code (-1) for the next node to destination to broadcast all such a packet to

all neighbours. The latter is enabled when the route tree is constructed, the parent of sender/forwarder is specified as a next node to indicate the forwarding packet via route.

6.4 Fault Model Implementation

As MintRoute constructs the route tree based on the link quality, we inject the fault model to validate our formal model. For example, we introduce the failed link between some nodes after the protocol has been active for some time. This section describes this fault model implementation. As we implemented the protocol communication and packet transmission models through refinement models *M0-M2*, we extend our developed formal model by implementing link failure events from these refinement models. We do not consider models *M3-M5* as they only encode the functionality of MintRoute protocol through refinement. The extended refinement structure for this fault model can be seen in Figure 6.26.

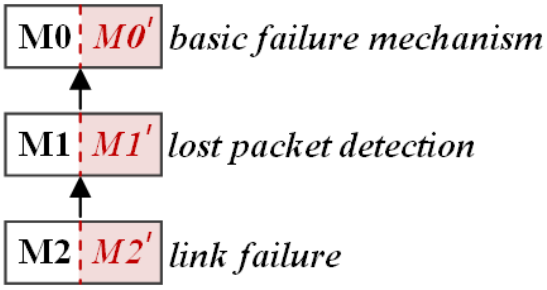


Figure 6.26: The extended refinement structure for fault models

6.4.1 Extended Initial Model (M0′) - basic communication failure

Initially, we start with modelling the very simple communication failure in which a packet transmitted to middleware can possibly be lost. Thus, variable *lostPkts* is defined in this refinement model to illustrate a set of lost packet ($lostPkts \subseteq PKT$). Event *net_losing* is defined to illustrate the packet lost on the network path. The lost packet is noticed in variable *lostPkts* and removed from the *middleware*.

6.4.2 Extended First Refinement Model (M1′) - lost packet detection

In this extended first refinement model, we introduce more concrete communication failure. As the neighbour node is introduced in this refinement, we define variable *recLostPkts* to indicate the lost packet recorded by each node ($recLostPkts \in ND \leftrightarrow PKT$). This variable is also a gluing invariant of abstract variable *lostPkts* ($ran(recLostPkts) = lostPkts$). We refine event *lose_pkt* from abstract event *net_losing* to perform this lost packet detection mechanism.

6.4.3 Extended Second Refinement Model (M2′) - link failure and recovery

This refinement model implements the link failure and recovery. To demonstrate this, two new events *disconnect_link* and *recover_link* are implemented. The first event assumes the link to be failed by removing the link from the dynamic wireless network (*wsnLinks*), whereas the second event recovers the failed link by adding such a link back to the wireless network.

Furthermore, this refinement model also tackles the situation when the sender node has broken links to all neighbours. We define event *lose_all_neighbours* which enables when the sender has no neighbours as all surrounding links are broken. This will indicate *FAILED_XMIT* to variable *envNeighbours* as shown in action *@a2* in Figure 6.27(a). Sequentially, this value will be updated in variable *ctlNeighbours* by event *sent_up* (Figure 6.27(b)). Finally, the lost packet will be detected by event *lose_pkt* as expressed in guard *@g4* in (Figure 6.27(c)).

```
(a)
event lose_all_neighbours
  any s nbs f pkt
  where
    ...
    @g4 f ↦ pkt ∈ channel
    @g5 nbs = wsnLinks[{f}] ∧ nbs = ∅
then
    @a2 envNeighbours ≔ envNeighbours
                ∪ {pkt ↦ FAILED_XMIT}
end
```

```
(b)
event send_up
 refines indicate_neighbours
  any pkt nbrs f sno src fwdr dat nbh
  where
   ...
   @g5 ran({pkt} ◁ envNeighbours) ∈ (ℙ(ND)
                ∪ {{FAILED_XMIT}})
  then
   @a1 ctlNeighbours ≔ ctlNeighbours
                ∪({pkt} × nbrs)
   ...
  end
```

```
(c)
event lose_pkt refines lose_pkt
 any f pkt
 where
  ...
  @g3 f ↦ pkt ∉ recLostPkts
  @g4 ran({pkt} ◁ ctlNeighbours) =
              {FAILED_XMIT}
 then
  @a1 recLostPkts ≔ recLostPkts ∪ {f ↦ pkt}
  ...
end
```

Figure 6.27: Events defined in the extended second refinement illustrating losing all neighbour surrounding the sender

6.5 Event-B Development Discussion

In the earlier section, the Event-B model of our WSN system was discussed through refinement steps. We have discovered that the refinement and its atomicity technique in Event-B influence the benefit to our WSN protocol development in terms of separation of concerns. They enable us to manage the complexity in S-style development which causes the monolithic model development at each layer of protocol stack. At each layer implementation, all functionality corresponding to that layer protocol algorithm are required to be integrated with the standard library provided by simulation toolkit into a single image. For example, in the network layer of our simulation model, we had to integrate all MintRoute functionality (e.g. link quality estimation, route broadcast and parent selection mechanisms in MintRoute) with the MiXiM's standard interface (e.g. gate operations implementing sending/receiving between layers and packet encapsulation/decapsulation) into a single model. The use of refinement and its atomicity provided by Event-B enables functionality of a single protocol algorithm to be layered through refinement steps as shown in Table 6.3.

Table 6.3: Refinement steps and their corresponding requirements

	Refinement Steps	**Descriptions**	**Corresponding requirements**
SensorScope	Initial model (M0)	Basic communication protocol	**FUN-1, FUN-2, ENV-2**
	First refinement (M1)	Neighbour nodes and sensing mechanism	**FUN-1b**
	Second refinement (M2)	Network topology, physical environment and channel mechanism	**FUN-1a, ENV-1**
MintRoute	Third refinement (M3)	The neighbourhood discovery and link quality estimation mechanism of MintRoute algorithm	**REL-1, FUN-3a, FUN-3b**
	Fourth refinement (M4)	Route broadcast mechanism	**REL-1(REL-1a, REL-1b), FUN-3c, FUN-3f**
	Fifth refinement (M5)	Parent selection and unicast packet forwarding mechanism	**FUN-3d, FUN-3e, SAF-1, SAF-2(SAF2.1-SAF-2.2)**

Furthermore, We have found that Event-B cannot analyse and evaluate the performance issue as well as simulation does. Hench, performance requirements **PER-1** and **PER-2** are demonstrated and evaluated in the simulation experiments (as mentioned in the previous chapter).

As mentioned in the earlier section that we focus on the Event-B development of two upper layer protocols, SensorApp and MintRoute, we have not modelled **FUN-4** (including **FUN-4a**, **FUN-4b** and **FUN-4c**) and **REL-2** regarding S-MAC protocol as explained in Section 4.6 and 4.7 respectively. We retain S-MAC model, achieved from S-style development as described in Chapter 5, in MiXiM framework. This will be used to demonstrate the flexibility of the iterative co-model development and co-simulation

between the node controller model in Event-B and the physical simulation model in MiXiM in the next chapter.

6.6 Model Verification and Validation

During development of the WSN model, we also verified and validated our model. The goal of model verification and validation is to verify whether the model fully satisfies all expected requirements and safety properties. The effective forms of verification and validation techniques, namely POs and animation are used to verify and validate our WSN model.

6.6.1 Model Verification by POs

We verify the properties of a machine by proving that every event still satisfies invariants. 330 POs were discharged, in which 85% of the total number of POs were proved automatically by RODIN. This includes the invariant to guarantee that no packets are lost under the perfect network. Furthermore, the safety invariant regarding the absence of loop problem was discharged automatically. However, the remaining were proved interactively. This is because invariants and *tcl* properties include quantified predicates and graph properties.

6.6.2 Model Animation and Validation

We create the testing scenarios concerning the different link quality ratios in order to exercise all desired requirements and strategies. These are used to animate and trace the list of operations to validate the formal model on ProB. An animator, ProB, is the tool that provides a model visualization environment to check the correct animation of traces of the Event-B model. The list of events can be traced step by step in which the state of the machine can be verified as to whether there are violations to the invariant.

To validate our network model behaviours, firstly, we create the basic scenarios (as shown in **Scenario 1**, Table 6.4) that satisfy all desired requirements and strategies, before the fault scenarios are created. Then, we manually execute the sequence of events corresponding to such fault scenarios. The state of the variable is also monitored, while animating the sequence of events. The model will be modified, if there are unexpected and undesirable behaviours. These scenarios are summarised as shown in Table 6.4.

Scenario 2: since the calculation of number of lost packets is a prerequisite operation that affects to the correctness of the later operations e.g. link quality estimation and parent selection, we perform specific testing and validation of this calculation. As explained

Table 6.4: Summary of testing scenarios used in animation validation

Scenario Id	Descriptions	Objective
Functional testing and validation		
Scenario 1	No Packet loss	To test and validate the correctness of packet transmission from SensorApp protocol and MintRoute functionality and algorithm
Failure testing and validation		
Scenario 2	Introducing packet loss from link/node failure	To enable specific testing and validation at calculation of the number of lost packet
• **Scenario 2-1:** losing the first beacon at the begining period *t* • **Scenario 2-2:** losing one beacon in the middle of period *t* • **Scenario 2-3**: losing multiple beacons in the middle of period *t* • **Scenario 2-4:** losing the last beacon at the final period *t*		
Scenario 3	Introducing link failures	To test and validate the correctness of parent selection mechanism in MintRoute model
Scenario 4	Introducing node failures	To test and validate the survivability of the network of parent selection mechanism in MintRoute model.

in Section 6.3.5, event *update_nbr* calculates the number of lost beacons transmitted between node x and y ($missed_{x-y}$). We expand this calculation into four scenarios as shown in Table 6.5. In this table, the columns from left to right represent: firstly, the scenario with its description, secondly, an example, thirdly, the calculated number of lost packet (*missed*), fourthly, the number of received packet (*received*) and lastly, the sequence number of the last received beacon (*lastSeqno*). This table assumes that node x receives a beacon from node y and there are five beacons transmitted during the link quality estimation at time t. In column examples, symbol ✓ denotes the previous sequence number received at node x, symbol ✗ denotes the sequence number in lost packet and symbol ↓ denotes the current received sequence number.

Table 6.5: Four testing scenarios for the strategy of calculating the number of lost beacons

Scenarios	Examples	miss(x ↦ y)	received(x ↦ y)	lastSeqno(x ↦ y)
Scenario 2-1: *losing the first beacon at the beginning of period (t-1,t)*	*t-1* *t* ✗ ↓ **Seqno:** 1 2 3 4 5	2-(0+1) = 1	1	2
Scenario 2-2: *losing one beacon in the middle of period (t-1,t)*	✓ ✗ ↓ **Seqno:** 1 2 3 4 5	3-(1+1) = 1	2	3
Scenario 2-3: *losing multiple beacons in the middle of period (t-1,t)*	✓ ✗ ✗ ✗ ↓ **Seqno:** 1 2 3 4 5	5-(1+1) = 3	2	5
Scenario 2-4: *losing the last beacon in the period (t-1,t)*	✓ ✓ ✓ ✓ ✗ **Seqno:** 1 2 3 4 5	4-(3+1) = 0*	4*	4*

Considering interesting *scenario 2-4* in this table as we mark with asterisk (*), we introduce the last beacon from node y to x to be lost during estimation time interval *(t-1,t)*. In this case, there is no calculation of the number of lost beacons. The value of fields received and *lastSeqno* of node y recorded in the neighbour table of node x is the value calculated from the previous calculation (when this node receives the beacon that has sequence number 4 in this example). This means that this lost packet is not captured by event *update_nbr*. However, since each sender/forwarder node records the number of actual transmitted beacons within the link quality estimation time interval *(t-1,t)* (in this case 5 beacons). The number of lost packet in this case is revealed in the link quality estimation step.

Scenario 3: This scenario aims to validate the parent selection mechanism. We introduce the failed links between some nodes after the packets had been transmitted for a while (in Figure 6.28(b)) to drop the quality of those failed links. Consequently, at the end of animation, such low quality links are not considered as routes for transmission in the parent selection process as shown in 6.28(c).

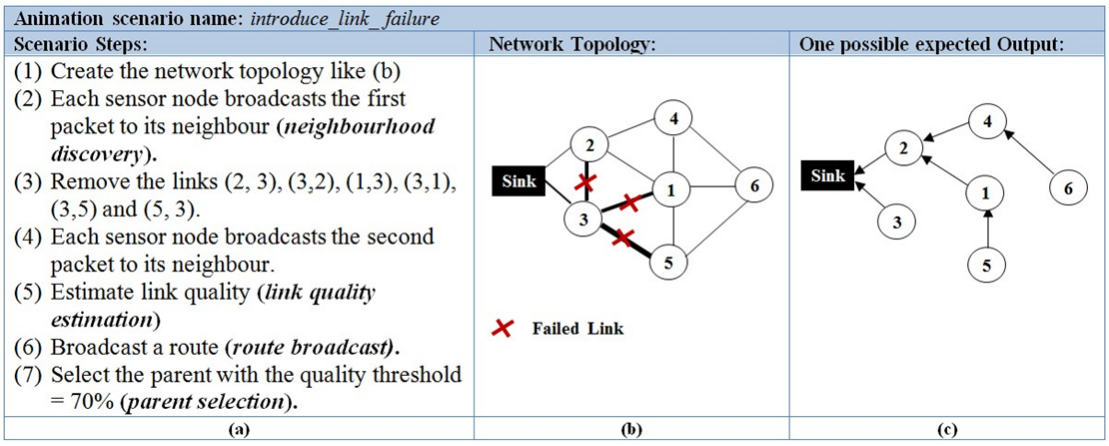


Figure 6.28: Fault scenario *introduce_link_failure* for network model animation and its result

Scenario 4: Another example of fault scenarios is shown in Figure 6.29. The objective of this scenario is to validate the survivability of the network in which the network must be connected when injecting the dead node. In this scenario, we indicated node 3 to be disconnected from the network by removing all corresponding links surrounding this node as shown in Figure 6.29(a-b). We found that the remaining alive nodes can reach the sink in the route tree as illustrated in Figure 6.29(c).

6.7 Model Decomposition

As WSN is a distributed system in which each node exchanges data via a channel interface, shared event decomposition (Butler, 2009; Silva et al., 2011) is used to separate

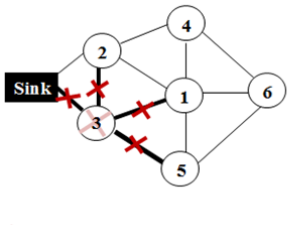
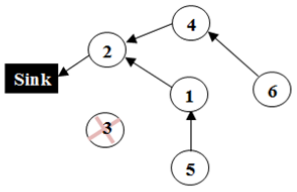
Animation scenario name: *introduce_node_death*		
Scenario Steps:	**Network Topology:**	**One possible expected Output:**
(1) Create the network topology like (b) (2) Each sensor node broadcasts the first packet to its neighbour (***neighbourhood discovery***). (3) Inject node 3 to be dead by removing all corresponding links surrounding node 3. (4) Each sensor node broadcasts the second packet to its neighbour. (5) Estimate link quality (***link quality estimation***) (6) Broadcast a route (***route broadcast***). (7) Select the parent with the quality threshold = 70% (***parent selection***).		
(a)	**(b)**	**(c)**

Figure 6.29: Fault scenario *introduce_node_death* for network model animation and its result

software controller from environment as shown in Figure 6.30. The software controller contains the necessary variables and events expressing communication networking protocols and algorithms such as SensorApp and MintRoute whereas environment consists of variables and events regarding the connectivity, channel and sensing environment. We design these two subcomponents to exchange messages via shared events. The steps of communication can be described as follows:

(P1) The environment activates each sensor node in the controller to sense and create a data packet via a shared event (*sensing*).

(P2) Each sensor node transmits a data packet down to the channel. The shared event (*send_down*) passes the forwarder node id and packet information as channel parameters (*chnPar*).

(P3) The channel indicates the neighbour of the current forwarder.

(P4) The channel returns the neighbour list and the forwarded packet to every neighbour controller via the shared event (*send_up*) to indicate the operation transmitting a packet up to specific receivers.

(P5) Each neighbour node (including a sink, e.g. nodes 0, 1, 4, 3 of sender node 2, Figure 6.30) receives a forwarded packet (*receive_pkt* and *sink_recv_pkt*) or detects a duplicated packet (*receive_dup_pkt*).

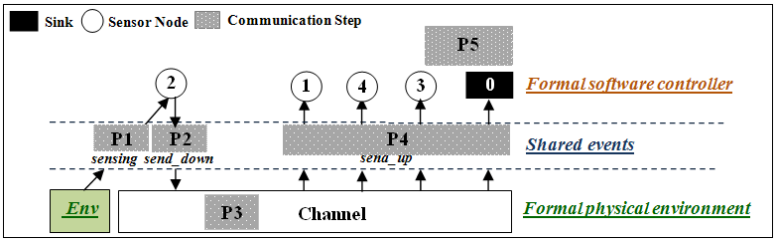


Figure 6.30: F-style co-simualtion steps for SensorApp protocol

6.8 F-style Co-simulation

In section 6.6.2, we demonstrate the validation of our developed formal model by using animator ProB in which the state of the machine is traced manually step by step. Since ProB 2.0 provides APIs that can access the states and events in Event-B machine as described in Section 2.8.3, this section demonstrates the automatic validation of the Event-B model by using these APIs. Another aim of this section is to prepare and validate this executable formal controller which we deploy in our C-style (*FoCoSim-WSN*) framework explained in the next chapter.

We use the interaction between our separated controller and environment models achieved from the decomposition step as explained in the previous section to drive our formal simulation. To accomplish this, a master is developed by using Groovy scripts. This master is responsible for co-ordinating and co-simulating between the controller and environment models. We exercise this F-style simulation at two levels SensorApp and MintRoute as shown in Figure 6.31. Based on the refinement structure described in Figure 6.2, machines *M2* and *M5* are the final model of SensorApp and MintRoute respectively. Thus, the shared-event decomposition technique is applied at these levels. Figure 6.31(a) and (b) show the result of SensorApp and MintRoute model decomposition respectively. The separated controller of these two layer protocols interact with their corresponding environment model by using shared events.

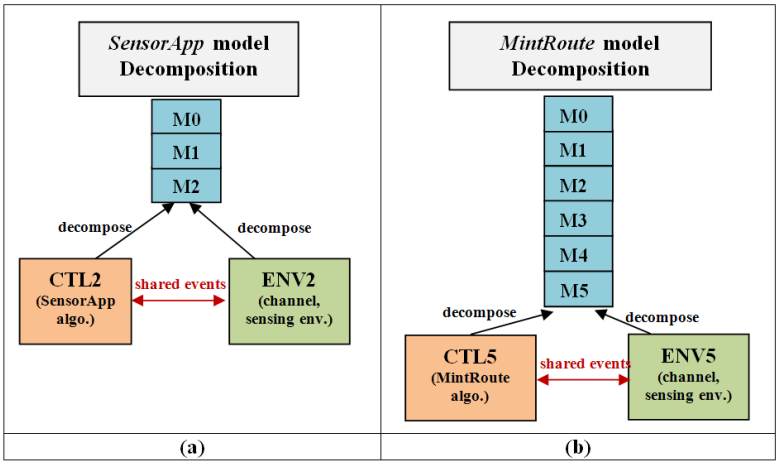
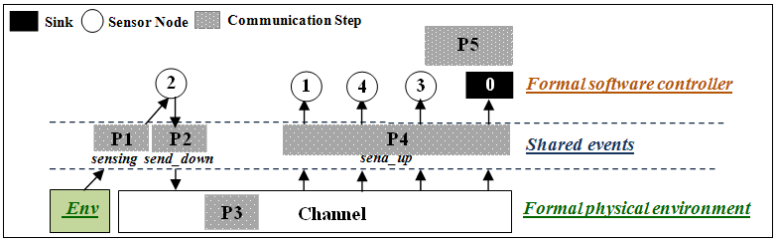


Figure 6.31: the decomposition of (a) SensorApp and (b) MintRoute models

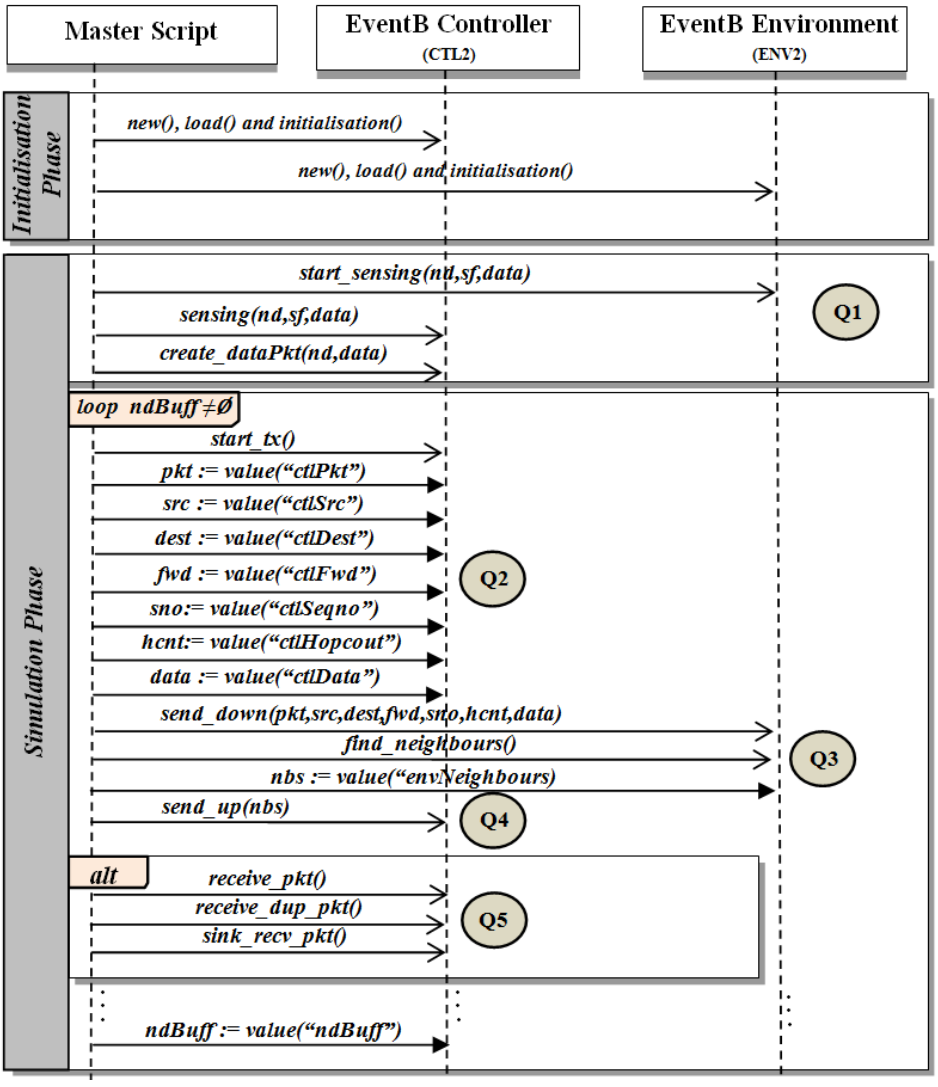
Figure 6.32(b) shows the diagram of the master algorithm of F-style simulation. The sequence of steps in this figure corresponds to steps expressed in Figure 6.30 (The copy of this figure is shown in Figure 6.32(a)). Step *Q1* expresses the sensing operation in which the environment initiates each node (*nd*) to sense data (*data*) by using event *start_sensing*. Each node senses data from environment by using shared event *sensing* (as demonstrated (*P1*), Figure 6.30) and then attaches such data into a created data packet (event *create_DataPkt*, Figure 6.32(b)).

Considering step *Q2* which corresponds to step *P2* in Figure 6.30, each node retrieves a packet in a waiting buffer by using event *start_tx*. In this step, the master prepares the parameters transmitted to the environment model. These parameters are extracted from the selected packet including necessary fields source (*src*), destination (*des*), forwarder (*fwd*), sequence number (*sno*), hop count (*hcnt*) and data (*data*). Shared event *send_down* synchronise these parameters to the environment model. Step *Q3* (corresponding to *P3*, Figure 6.30) indicates the neighbour list of the forwarder by event *find_neighbours* before this neighbour list is returned back to the controller by shared event *send_up* as shown in step *Q4* (corresponding to *P4*, Figure 6.30). Step *Q5* (corresponding to *P5*, Figure 6.30) demonstrates the receiving mechanism for the receiving node to receive the forwarded packet, method `anyEvent` provided by ProB Groovy is used to create alternative choices among events *receive_pkt*, *receive_dup_pkt* and *sink_recv_pkt*.

Figure 6.33 shows an example of master execution results at application and network layers. Figure 6.33(a) illustrates the partial execution result of broadcasting packet mechanism by forwarder node 2 in which packet PKT11 created by node 1 is broadcast to all node 2's neighbours. These neighbours are specified by the environment model and return them as a list to the controller. After transmission, packet PKT11 is received by a sink and detected as a duplicated packet by nodes 1 (as its own packet), 3 and 4 (as received it already). Figure 6.32(b) shows the route tree created by MintRoute model. This route tree corresponds to one possible route tree created by S-style model described in the previous chapter.



(a)



(b)

Figure 6.32: (a) Copy of Figure 6.30: F-style co-simualtion steps for SensorApp protocol and (b) F-style master algorithm for SensorApp protocol co-simulation

	F-Style Co-simulation	
SensorApp (M2)		Controller: node 2 forwards a packet PKT11
	Q2	Controller: sending channel information ctlPkt=PKT11, ctlSrc=1, ctlDest=-1,ctlFwdr=2,ctlSeqNo=1, ctlHopCount=0,ctlData=12.34
	Q3	Environment: receiving channel information: envPkt=PKT11, envSrc=1, envDest=-1,envFwdr=2,envSeqNo=1, envHopCount=0,envData=12.34 Environment: finding node 2's neighbours Environment: sending node 2's neighbours {0,1,3,4} …
	Q4	Controller: received neighbours {0,1,3,4}
	Q5	Controller: receiving information:: received ={} duplicated ={1,3,4}. Sink received={1}. Sink received packet PKT11

(a)

MintRoute (M5)	Controller: finished MintRoute algorithm for F-style simulation Controller: route tree= {(1|->2),(2|->0),(3|->0),(4|->2),(5|->1),(6|->1)}

(b)

Figure 6.33: Master execution result of F-style simulation

6.9 Comparison of S-style simulation, Model Animation and F-style co-simulation

We discover that ProB animation provided the microscopic view of system behaviours in which a single-step manual event invocation is created. It leads us to input the data manually for tracing the model behaviours closer, interacting with the environment step by step. States of variables and invariants are inspected as to whether they satisfy the functional requirement. Unlike ProB animation, simulation (both in F-style and S-style) enables the macroscopic view of model execution. The large-scale scenario for long running is generated automatically by simulation environments. However, F-style has no simulated real physical environment which is required in order to evaluate the non-functional requirement such as network performance.

Furthermore, we also observe informally that the MiXiM environment in S-style appears to trace-refine the Event-B environment model in F-style. For example, the connection between each node is implemented abstractly as the static configuration in the environment model to form up the network topology. Each node connects to each other

abstractly away from the real physical environment. Unlike this, the physical environments such as wireless channel, radio and analogue models provided by MiXiM are more concrete and realistic. The network topology is generated based on the maximal interference distance between nodes. As a result of this, simulation can strengthen the limitation of this F-style development in which the verified formal models representing the protocol algorithm can be run with the long running simulation under the realistic physical environment. In the next chapter, we integrate and drive this executable formal controller model with MiXiMs environment under our co-simulation framework.

6.10 Conclusion

In this chapter, we have demonstrated the value of formal modelling and analysis for WSN development using the SensorScope case study. Firstly, refinement techniques provided by Event-B language implemented each single controller for each protocol stack in a simulation model into multiple layers. This increases the understanding of the system and reduces the complexity in development. This confirms sub-contribution C.1 explained in Chapter 1 in which refinement and decomposition techniques provide high-level abstraction and separation of concerns.

Furthermore, the absence of certain classes of fault such as the looping problem in a route tree can be guaranteed. Certain requirements and safety properties have been encoded in invariants. The invariant for the no-loop property has been proved. Using the animation technology like ProB creates primitive validation in which the confidence in the formal model is gained by animating the controller model with the environment model step by step through various testing scenarios. To analyse and evaluate the performance of the network through long running simulation, shared event decomposition is applied to separate the formal controller model from the environment. The separated controller model is finally integrated with a realistic simulated environment provided by WSN simulation as described in the next chapter. As a result of this, this chapter has shown clearly that value can be added to simulation by formal modelling and analysis. From the result of this chapter and Chapter 5, it clearly shows that proof-based formal verification and simulation-based testing techniques can compliment each others' weaknesses. Therefore, the next chapter will demonstrate sub-contribution C.1 which is the integration between proof-based formal method like Event-B with MiXiM simulation-based development.

Chapter 7

Co-simulation Development (C-style Modelling)

7.1 Introduction

In Chapter 1, we proposed our main research contribution that is a formal co-simulation framework. This framework provides an integrated set of methodologies for WSNs: (S)imulations, (F)ormal and (C)osimulation which enables an iterative and incremental interworking scheme between the formal node controller and the physical simulation environment through multiple refinement levels shown in Figure 7.1. In F-style development, the protocol algorithm at each protocol stack layer is formally developed and verified through refinement steps. S-style development, on the other hand, enables the modelling of informal physical environment e.g. stochastic sensed data, channel and radio environment, allowing the simulation scenarios to be defined. The verified controller model is co-simulated with the environment model in C-style framework to execute the performance analysis.

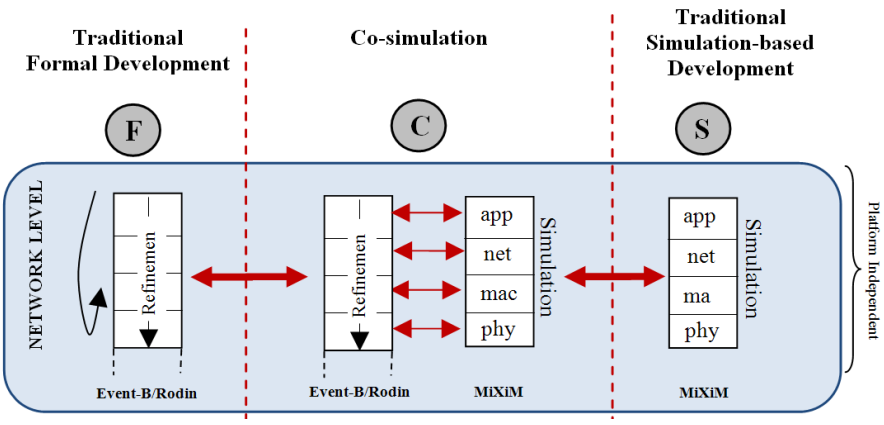


Figure 7.1: Vision of co-simulation approach for WSN development

Consequently, this chapter describes a developed prototype Formal Co-simulation (*FoCoSim-WSN*) framework, which provides co-modelling and co-simulation for WSN development. This framework is based on a formal co-simulation method for Event-B and MiXiM for WSNs which provides multiple abstraction levels for co-simulation through refinement. This results in the iterative and incremental development which distributes the functionality of WSN systems including protocol algorithms between components. This gives flexibility in WSN development though the iterative interworking and integration scheme of S-, F-, and C-style development as mentioned above. This enables engineers to be able to perform either S- or F-style development before the separated controller and environment are combined and co-simulated in this simulation framework. This means that the system models can be implemented easily to cross over to these three flexible modes of working.

Section 7.2 provides an overview and literature reviews of co-simulation approaches. In Section 7.3 and 7.4, we describe a general prototype *FoCoSim-WSN* architecture together with its implementation. The simulation steps for our formal co-simulation framework are described in Section 7.5. In Section 7.6, we demonstrate the use of this framework to implement the specific case study. This section also demonstrates how our framework enables the level of abstraction of development in which the functionality of SensorScope case study is distributed between the component. Section 7.7 describes the key programming constructs used to define master scripts. This section also explains an example of the developed master algorithm used to drive co-simulation. Section 7.8 describes the validation of this C-style framework compared with that of S-style. Section 7.9 describes the flexibility of co-simulation development. This section demonstrates two styles of interworking evolution: *from-S-through-F-to-C* (S^F *Co-simulation*) and *from-F-through-S-to-C* (F^S *Co-simulation*) development. These can be used as a guideline for potential co-simulation development. The details of the implementation effort required to develop our co-simulation framework are also provided in Section 7.10. Section 7.11 demonstrates the conclusion of this chapter.

7.2 Co-simulation Approaches

Co-simulation enables the partitioning of development, resulting in different subsystems representing a coupled problem being modelled and simulated in a distributed manner. Co-simulation also eliminates the limitation of the monolithic approach in which a single simulation model is developed under the constraint of a single simulation tool. It enables more efficient development in which specialised simulation tools for different domains can be used to develop coupled subsystem models before they are combined in a co-simulation environment.

Recently, co-simulation frameworks have been proposed in order to co-ordinate between the software controller model and physical simulation environment. DESTECS[1] (Design Support and Tooling for Embedded Control Software) (Broenink et al., 2010; Pierce et al., 2012) is an integrative co-simulation framework for co-modelling and co-simulating between discrete-event (DE) models of the control software and continuous-time (CT) models of the physical environment. In this co-simulation framework, the software controller, describing the systems functionality and dynamic behaviour, is modelled formally by VDM. 20-sim, on the other hand, is used to implement the differential equations of the physical environment model. The DESTECS tool allows these two models to be co-simulated and communicated with each other via XML-RPC Interface. The formal co-simulation of ADVANCE[2] (ADVANCE, 2013) provides a framework for integrating multi cyber-physical systems using different simulation engines via Functional Mock-up Interfaces (FMIs)[3]. A master-slave algorithm to execute the co-simulation is implemented on the ProB tool by using the Groovy - Java based language.

This research reduces the complexity of WSN development by separating the node controller containing protocol algorithms from the physical environment including communication channel, network topology and physical analogue and radio models. We apply a master-slave concept to integrate and co-simulate these two models. A master acts as a co-ordinator located between its slaves, which are the single simulation tools. The master algorithm is implemented to synchronise, control and manage these simulation tools. Figure 7.2 demonstrates our co-simulation concept. The master is implemented to co-simulate between two different simulation engines. One is ProB 2.0 - an integrated tool on Rodin that drives Event-B controller. The other is MiXiM that drives the physical environment models. As can be seen in this figure, the node controller in Event-B represents the protocol algorithm implemented in each sensor node e.g. the SensorApp and MintRoute algorithms, whereas MiXiM's physical environment models provide communication channel, network topology, physical and radio models. Since ProB 2.0 provides new APIs that facilitates the Groovy script language to cooperate with Event-B API to create the event trace of Event-B model, we develop the master algorithm on this ProB 2.0 by using this script language. This master implements the communication steps that are used to control the interaction between ProB and MiXiM simulators. Furthermore, input and output parameter lists (e.g. packet information and neighbour list respectively as expressed in Figure 7.2) are synchronised between these two simulation engines through communication steps. The next section demonstrates the parameter exchange in detail.

[1]see DESTECS http://www.destecs.org/
[2]see ADVANCE http://www.advance-ict.eu/
[3]see FMIs -https://www.fmi-standard.org/

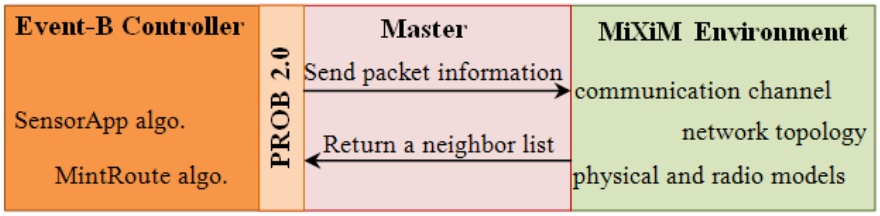


Figure 7.2: Vision of co-simulation approach for WSN development

7.3 *FoCoSim-WSN* Architecture for WSNs

This section demonstrates the conceptual co-simulation architecture that establishes overall structure of our co-simulation framework. It encompasses three components: *Event-B controller*, *Master* and *MiXiM environment* as shown in Figure 7.3.

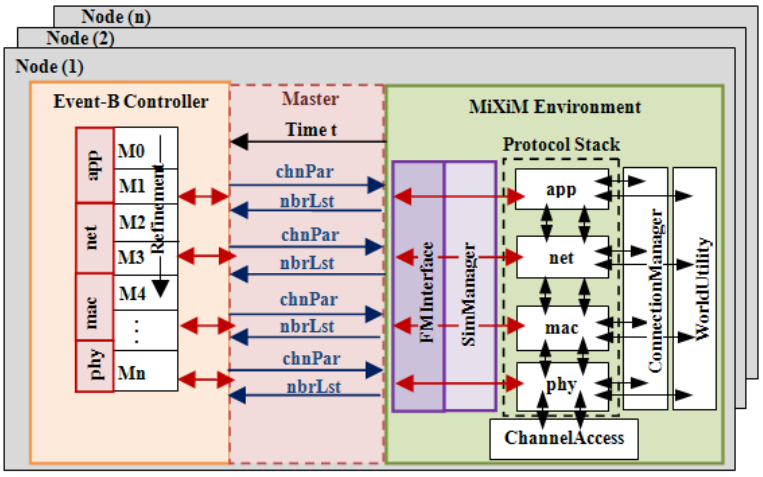


Figure 7.3: *FoCoSim-WSN* Co-simulation Architecture

(1) Event-B controller modelling: the node modelling controller, representing the functional requirements, computation and algorithm for each protocol layer is developed and verified through refinement steps in Event-B and its Rodin toolkit. Safety properties defined as safety invariants guaranteeing the absence of certain classes of fault. These POs are discharged as far as possible automatically by proof tools.

(2) MiXiM Environment Development: MiXiM provides physical and communication environment for formal node controller models. MiXiM's virtual node encodes the base protocol stack containing the gates for the communication between formal models and environment e.g. physical and wireless channel models. The generated virtual nodes are placed on the simulation area (playground) generated by module `WorldUtility`. The connection between these nodes is created by `ConnectionManager`. Only nodes that are placed within the maximal interference distance can communicate to each other. This is used to identify who is the neighbour to receive a transmitted packet from the

sender. In MiXiM, module `ChannelAccess` provides the mechanism to transmit a message transmitted from physical layer models to a channel.

Furthermore, we develop `FMInterface` as a mocked-up front-end interface in MiXiM that contains the synchronisation events corresponding to shared-events defined in Event-B. This synchronisation event in MiXiM is scheduled at fixed intervals for parameter synchronization exchange between the Event-B controller and MiXiM environment. `SimManager` is the back-end interface where the parameters passed from `FMInterface` are transited down to/up from the channel (accessed by module `ChannelAccess`) via the protocol stack. Each protocol layer in MiXiM environments only contains gates (without any protocol algorithms) for communicating with the adjacent layer.

(3) Master Algorithm: A master, Groovy script - programmable API for ProB, co-simulates between formal node controller and simulation environment. To enable our Event-B controller model to drive the packet transmission and protocol algorithm, the sequence of events is defined as an event trace in this controller model. These traces are coded in our developed Event-B interface (`EventBCtl`) in the master. This class enables an instance of Event-B model to be created and implemented as a thread in multithreaded master. Each thread represents the controller of each sensor node and exchanges input/output parameters with the physical and simulation environment encoded in the corresponding virtual node in the MiXiM. Periodic timers such as a sensing timer and route broadcast timer provided by MiXiM allows the Event-B controller and MiXiM environment models to exchange input/output parameters periodically. These parameters are exchanged via a TCP socket implemented in the master for Event-B controller and MiXiM Interface. This enables the input and output parameter to be exchanged between protocol algorithms and sensor environment. Parameter `chnPar` is defined as an input parameter containing the packet information required by MiXiM environment. This parameter is generated after the node controller creates a packet. Then, it is sent down to MiXiM environment. The output parameter `nbrLst` containing the receiving neighbour list is returned back from such simulation environment to the node controller. These parameters are encoded in XML formats and demonstrated in Appendix B.3. Note that parameter `chnPar` is designed based on the generic base message necessarily used for each protocol layer provided by MiXiM. Thus, the generic field for parameter `chnPar` consists of the initial source of packet (`cSrc`), the next node id (`cParent`, setting to -1 for broadcasting mechanism), the forwarder (`cForwarder`), the sequence number (`cSeqNo`), the hop count (`cHopCnt`) and data (`cData`) field. Data field is for specific use. For example, in our SensorScope development, we use this field to record sensed data for a data packet in application protocol and transmission link quality ratio for a route packet in routing protocol. However, the input/output parameters for co-simulation at MAC and physical layers are work in progress.

7.4 Co-simulation Implementation

In the previous section, we describe the co-simulation architecture which consists of three components: *Event-B controller*, *Master* and *MiXiM environment*. To establish our co-simulation framework, this section expresses the implementation level of this architecture corresponding to these components.

7.4.1 Event-B Controller

In Chapter 6, we demonstrated that two protocols at upper layers of protocol stack, SensorApp and MintRoute, are layered and verified through multiple refinement steps. To demonstrate the usability and flexibility of the co-simulation framework, we decompose the final refinement models at two upper layer protocol developments: *M2* for SensorApp, the protocol at application layer and *M5* for MintRoute, the protocol at network layer as described in Section 6.7. These two models, then, are driven by a master in order to co-simulate with MiXiM environment models. The shared-event provided by these two controller models are used to synchronise the input and output parameter during co-simulation. Shared events *sensing*, *send_down* and *send_up* are defined for synchronising sensing mechanism, passing parameter `chnPar` from a sender controller down to channel environment in MiXiM and returning parameter `nbrLst` from such channel environment up to the receiving controllers as demonstrated in Figure 7.4 respectively. The communication steps for C-style co-simulation demonstrated in this figure is designed based on the communication steps of F-style co-simulation shown in Figure 6.30 (A copy of this figure is shown in Figure 7.8(a)). We only replace a formal environment model with the MiXiM simulation model. The co-simulaion steps with the parameter synchronisation mechanism are described in the later section.

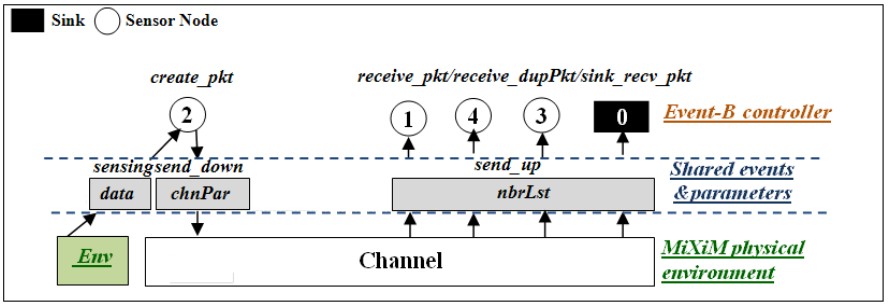


Figure 7.4: Shared events in Event-B controller used to synchronise input and output parameter to MiXiM environment corresponding to Figure 6.30

7.4.2 Master Implementation

As mentioned in the previous section, a master is a co-ordinating component that co-simulates between Event-B controller and MiXiM simulation models. It aims to drive

this formal controller by creating the event trace to demonstrate the WSN behaviour e.g. the packet transmission and protocol algorithms. Furthermore, this master also accesses the state variable of this controller in order to generate the necessary input and output exchanged with simulation environment.

As one of our research sub-contributions is to provide the reusable framework which can be applied to other applications, our master structure is divided into six classes as demonstrated in Figure 7.5. Further details of these classes are given in Appendix B.1.

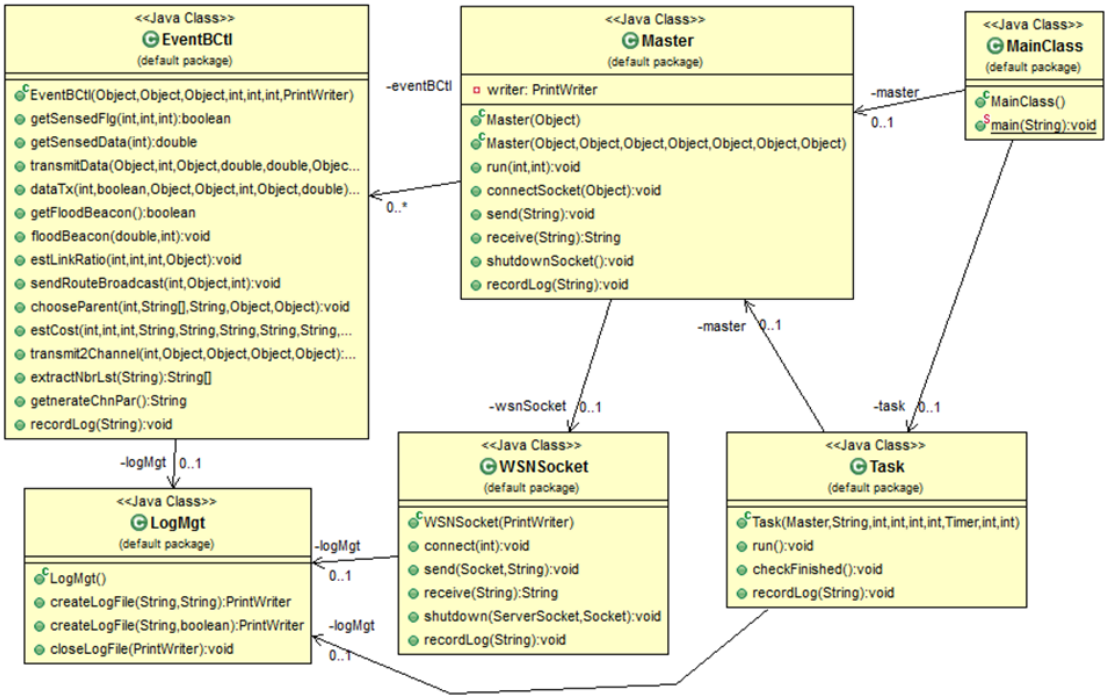


Figure 7.5: Master structure implementation

- **MainClass:** the main class drives the master execution on ProB. This class is responsible for initially establishing and checking the connection between the Event-B controller and MiXiM environment models. The parameters for initialization such as the number of nodes and the sink id configured in MiXiMs configuration file are passed to the Event-B model via this class. This class also creates multithread (represented in term "`Task`") and loads instance of Event-B model into each thread to illustrate each controller node. The scheduler for each thread (*`Task`*) is assigned in order to specify the periodic operation such as sensing for each node.

- **Task:** this class extends Java class *`TimerTask`*[4] in order to create threads and schedule the task for these threads periodically.

[4]in Java language, class *`TimerTask`* implements interface Runnable that creates instances executed by a thread. This class also contains a timer in which a task for a thread can be scheduled for particular one-time or repeated execution. For more details, see Oracle JavaDoc -http://docs.oracle.com/

- **Master:** this class is responsible for connecting to `EventBCtl` (Event-B controller interface class) and exchanging the input/output parameters generated by this class to simulation environment models via `WSNSocket`.

- **EventBCtl:** this is the controller interface class encoding the event traces in Event-B controller model including SensorApp and MintRoute models for SensorScope case study. This class also extracts the state of variables in order to generate the necessary input and output parameters exchanged to simulation environment. For example, method `getSensedFlg()` implemented in this class executes shared event *sensing* to get the sensed flag and data from the environment. Also, we implement method `transmitData()` to call shared events *send_up* and *send_down* to pass input and output parameters to simulation environment.

- **WSNSocket:** this class contains a socket interface (implementing Java class `ServerSocket` and `Socket`[5]) used to connect to MiXiM environment models.

- **LogMgt:** this class implements a Java output stream class to perform the log file operation. This includes the creation, open, write, and close operation on a file. The result of master execution is recorded in this log file.

7.4.3 MiXiM Extension Implementation

We extend the base module (the general structure) provided by MiXiM to implement the necessary interface for our formal co-simulation. The developed interfaces are divided into four modules (classes).

- **FMInterface:** as mentioned in the previous section, the main responsibility of this module is to synchronise the input/output parameter between formal controller models and simulation physical environments. Thus, this module contains TCP socket for the parameter exchange purpose. Input/output synchronisation methods `receiveChnPar()` and `sendNbrs()`, as shown in Figure 7.6, encode sending (`send()`) and receiving (`recv()`) operation implemented in C++ language . These methods are used to synchronise parameters with output and input shared-event *send_down* and *send_up* in Event-B controller respectively over a socket.

- **SimManager:** this module is a back-end interface that transits the input/output parameters passed by/to `FMInterface` down to/up from the channel via implemented simple modules `SimpleApp` and `SimpleNet`.

[5]class ServerSocket is used to implement a server socket, whereas classes Socket implements a client socket. See Oracle JavaDoc for more details -http://docs.oracle.com/

- **Simple modules SimpleApp and SimpleNet:** these modules are designed and created to be a base module for our co-simulation. Basically, they mainly provide the communication mechanism for communicating between the formal controller implemented at the upper layer of protocol stack and the adjacent lower layer. To accomplish this, we extend the simple modules from the existing base class named `BaseLayer`. This base class enables the communication layers of the protocol stack to be formed up. Therefore, two simple modules are layered. One is `SimpleApp` for application protocol layer, whereas another is `SimpleNet` for network protocol.

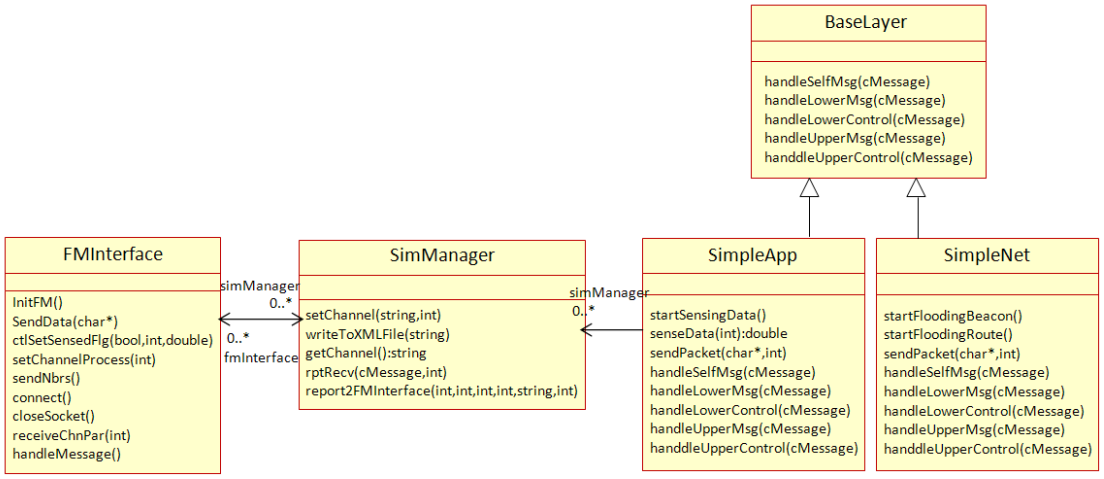


Figure 7.6: MiXiM's extension implementation

7.5 Co-simulation Steps

Figure 7.7 shows FoCoSim-WSN co-simulation steps which are classified into two main phases.

(I) Initialisation Phase: this starts with **(I1)** MiXiM initializing the physical environment such as interval time, ratio propagation, max transmission power and path loss coefficient from the configuration file. Then, the thread representing the virtual node in MiXiM is generated and the coordinate of each node in the configuration file is used to locate the corresponding generated virtual node on the simulation playground. The network topology is formed up by connectable virtual nodes (node placed within the maximal interference of each other). Next, **(I2)** MiXiM requests the connection to Event-B controller by using socket interface implemented in `FMInterface`. After this connection is established, `FMInterface` sends the necessary information such as the number of nodes and identified sink id extracted from the configuration file to a master. **(I3)** The master creates multiple threads for each Event-B node controller corresponding to the number of virtual node threads generated in the MiXiM environment. Note

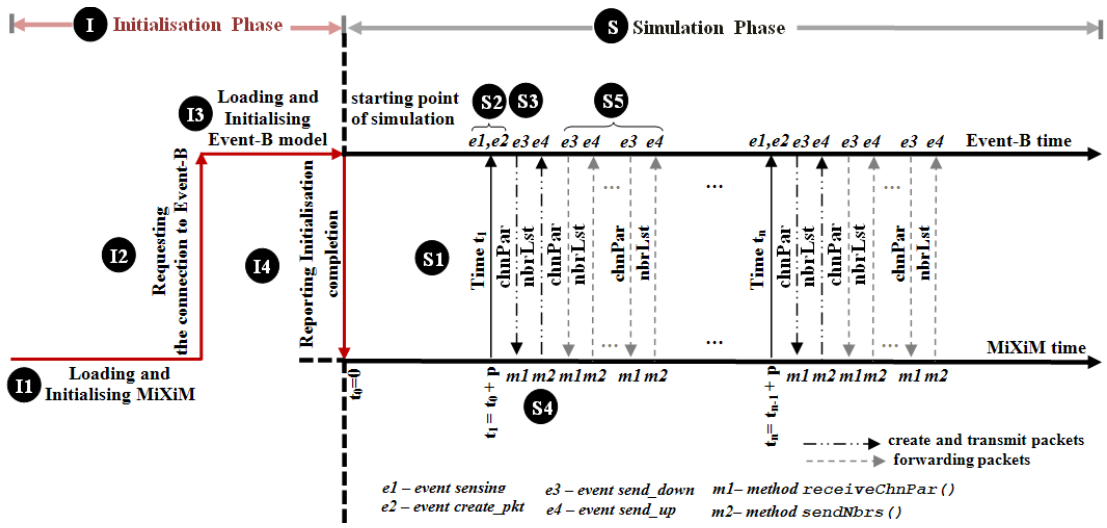


Figure 7.7: Master structure implementation

that, in MiXiM, we implement one thread per virtual node by using C++ as well as in Event-B in which one thread is implemented per node controller by Java Groovy in the master. Each thread of node controller in Event-B master is mapped onto each thread of virtual node in MiXiM environment. Then, the instance of Event-B controller is created, initialised and loaded in to each thread in the master. Finally, **(I4)** the master reports the completion on the Event-B model initialisation to `FMInterface` in MiXiM.

(S) Simulation Phase: the MiXiM begins to simulate at time 0 (*t0*). At each cycle timer *p* - e.g. **(S1)** for *t1*, MiXiM sends the triggered information to tell each controller node to start performing the operation such as sensing and creating a packet. **(S2)** the sense data is generated from MiXiM to the required random distribution for each node controller. This sense data is passed to the node controller by using shared event *e1* (e.g event *sensing* in Figure 7.4). Then, a packet containing such sensed data is created (by local event *e2*, e.g. *create_pkt*). **(S3)** next, parameter `chnPar` containing the packet information is synchronized between output shared event *e3* in Event-B controller (e.g. *send_down* in Figure 7.4) and the input synchronization method (*m1*, `receiveChnPar()` in Figure 7.6) in `FMInterface`. The virtual packet containing this corresponding input parameter is created and transmitted to the virtual neighbour node via the protocol stack. **(S4)** demonstrates receiving neighbour list `nbrLst` in MiXiM of the sender node being synchronized back from the output synchronization method (e.g. *m2*, `sendNbrs()`), Figure7.6) in `FMinterface` to input shared event *e4* in Event-B controller (e.g. event *send_up* in Figure 7.4) of each thread of node controller. After receiving the neighbour list, each node controller will perform the next operation (e.g the receiving events described in Figure 7.4). Then, **(S5)**, the receiving node that is not the destination forwards/rebroadcasts a received packet to its neighbours. The steps of this communication are the same as described for packet creation and transmission

mechanism. Steps **S1-S5** will repeat periodically until the simulation reaches the finish point.

Figure 7.8(b) shows the master algorithm interaction that is one of the concrete versions of co-simulation steps shown in Figure 7.7. This interaction demonstrates one sensing cycle for SensorApp corresponding to communication steps in Figure 6.30. As we use the same Event-B controller, we design steps R1, R2 and R4 to be the same as steps P1, P2 and P4 in Figure 6.30. The input parameter, that is the packet information (`chnPar`), is passed from shared event *send_down* in Event-B controller via a master to synchronised method *receiveChnPar()* implemented in MiXiM's `FMInterface` as demonstrated in step R2. Then, in step R4, the neighbour list, an output parameter, is returned by synchronised method `sendNbrs()` back to shared event *send_up* in Event-B controller. To exchange these parameters between these two simulation engines (a master in ProB and MiXiM), the socket methods *send()* and *recv()* are used in a Master and MiXiM's `FMinterface` for sending and receiving parameters respectively as shown in Steps R2 and R4. Step R5 is implemented for a receiving node to receive the forwarded data packet. Method `anyEvent` provided by ProB Groovy is used to create alternative choices among events *receive_pkt*, *receive_dup_pkt* or *sink_recv_pkt*.

Considering step R3 which corresponds to step P3 in Figure 6.30, after each node in MiXiM environment receives channel information from `FMInterface` via `SimManager`, the packet is generated based on this information and transmitted down to lower layers and finally a channel. Note that virtual nodes in the MiXiM environment are generated by module `SimpleApp` (the same as `SimpleNet` for the network layer) containing only gates for transmitting/receiving a data packet to/from lower layers. The forwarding node uses method `sendDown()` to relay a current packet down to the lower layer and finally to the channel. Method `handleLowerMsg()` is used by receiving nodes for receiving a forwarded packet from the lower layer. `SIMManager` collects information from receiving nodes to generate the neighbour list. Finally, this neighbour list is sent to `FMInterface` and relayed to the Event-B controller via the master.

Note that we also implement the master algorithm interaction for the MintRoute protocol to demonstrate the different level of abstraction in our co-simulation. However, we do not describe the master algorithm for the MintRoute protocol in this section. This is because it has the co-simulation interaction step to exchange the same input/output parameters for beacon and route packet transmission. The next section will describe the level of abstraction with a case study.

7.6 Co-simulation Case Study Modelling

This section describes how our FoCoSim-WSN framework enables the level of abstraction of development that distributes the functionality between the component. This section

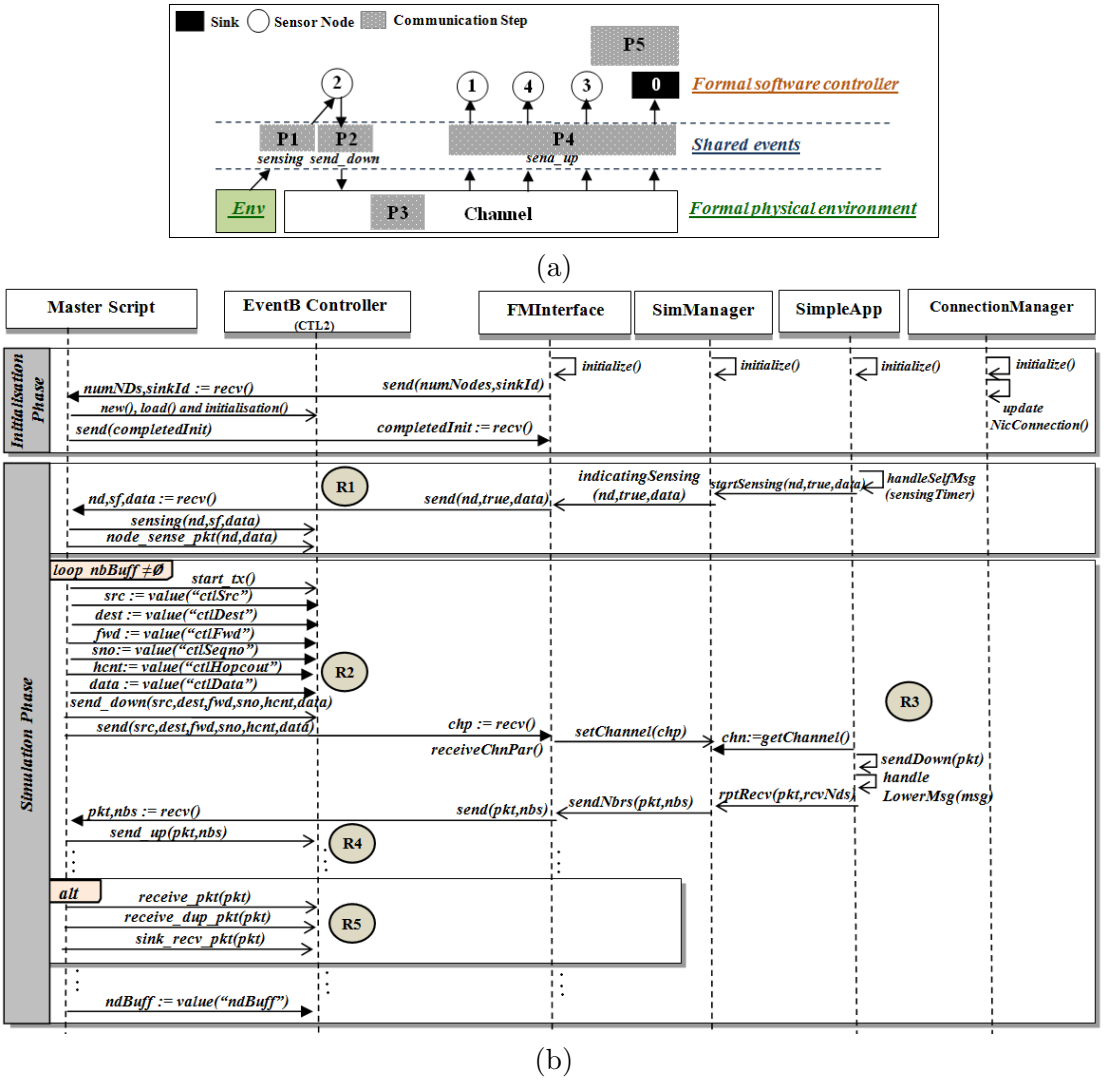


Figure 7.8: (a) Copy of Figure 6.30: F-style co-simualtion steps for SensorApp protocol and (b) C-style master algorithm for SensorApp protocol co-simulation

demonstrates three levels of abstraction in our SensorScope case study development: S-, $S^{Fo.2}$ and $S^{Fo.5}$ levels. We start with S-level development which has no partition of development. This level implements all functionality in MiXiM simulation. Then, the functionality of communication protocols is separated from S-level development and modelled in Event-B. We call this level of abstraction $S^{Fo.n}$ where n means the level of refinement in Event-B. Here, further levels of abstraction are described.

As demonstrated in the decomposition technique for F-style development (as expressed in Section 6.7), we decompose the final refinement model, representing the specific protocol algorithm for each layer of protocol stack, separately from environment model. We achieved two separated controller models. One is *CTL2* that implements SensorApp protocol for application layer. Another is *CTL5* that encodes the MintRoute protocol for network layer. Consequently, two further levels of abstraction are considered: $S^{Fo.2}$ and $S^{Fo.5}$ that correspond to the application and network layers respectively. We use

these separated node controller models in F-style development (as described in Section 6.8) to implement them in this mixed co-simulation. In the simulation environment, we develop gates for communication between the formal controller model at the upper layer with the environment model at the lower layer of protocol stack. This is accomplished by removing the protocol algorithms from SensorApp and MintRoute modules in S-style to implement two modules for upper layer protocol, `SimpleApp` and `SimpleNet`. These modules are a standard module extending the base modules in MiXiM containing only packet en/decapsulation functions and gates connected to the adjacent lower layer. This enables the corresponding Event-B controller representing the upper layer model to be able to communicate with the lower layer in MiXiM environment via `FMInterface` and `SimManager`. Our prototype co- simulation framework exercises these two levels of abstraction - application and network layers as shown in Figure 7.9(b) and (c) respectively.

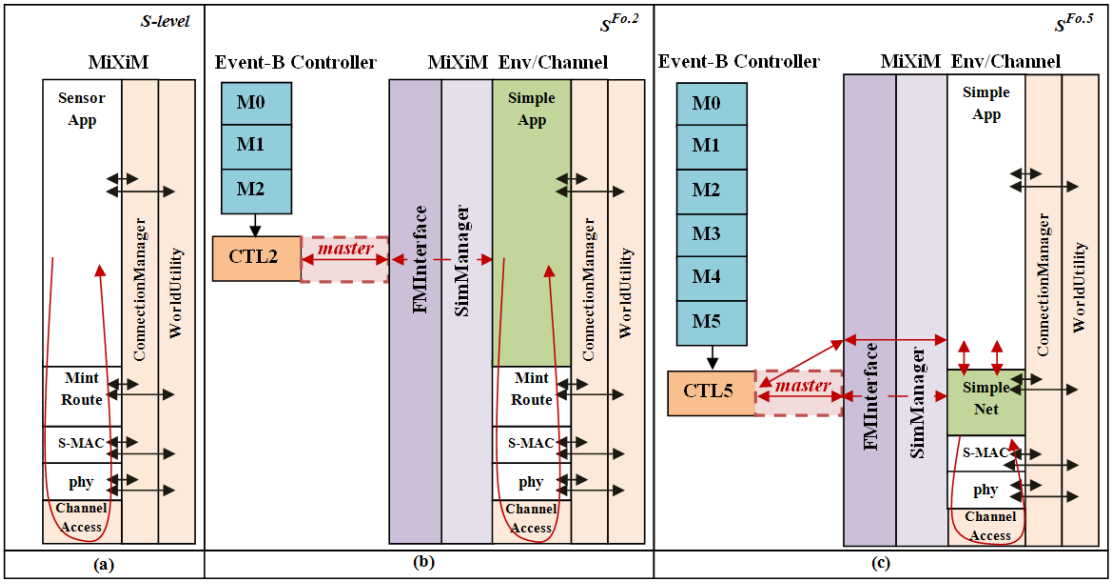


Figure 7.9: Co-modelling separating controller algorithms from (a) S-style model to (b) SensorApp co-models and (c) MintRoute co-models

To demonstrate the iterative co-model development for each layer protocol, we start from the achievement of S-style development to illustrate S-level of abstraction as shown in Figure 7.9(a). At each layer of protocol stack, the functionality of each protocol e.g. SensorApp, MintRoute and S-MAC is implemented together with the standard library function e.g. packet en/decapsulation functions and gates for connection between adjacent layers. Then, in Figure 7.9(b), we only separate the SensorApp algorithm from the simulation and implement in Event-B to enable the abstraction level of application protocol ($S^{Fo.2}$). The functionality of this separated SensorApp algorithm is layered though refinement steps as described already in the previous chapter. Then, the final model (*M2*) of this protocol development is decomposed from the formal environment named *CTL2*. This decomposed model is finally deployed into co-simulation framework

as shown in Figure 7.9(b). In MiXiM, module *SimpleApp* is used as a gate for communication between SensorApp controller in Event-B and lower layers retaining protocol algorithm MintRoute and S-MAC. Figure 7.9(c) shows the co-simulation at the network layer that provides the abstraction level of network protocol ($S^{Fo.5}$). The co-models for this layer are implemented in the same way as in the application layer. The final formal model of SensorApp is refined to implement the functionality of MintRoute protocol through multiple refinement steps. Similar to SensorApp development in Event-B, we deploy decomposed controller model, *CTL5*, to our co-simulation framework. Module *SimpleNet* in MiXiM is used to coordinate between this MintRoute controller in Event-B and the lower layer. S-MAC is still retained in MiXiM's MAC layer. As this Event-B controller also contains the concrete model of SensorApp protocol containing unicast data packet forwarding mechanism via the route, this model also co-simulates with module SimpleApp in MiXiM.

7.7 Master Scripts

In Section 7.5, we described co-simulation steps enabling the interaction between Event-B and MiXiM models driven by a master. This section explains the key programming constructs used to define master scripts together with demonstrating by using an example of a master algorithm.

7.7.1 Key Programming Constructs for Master Scripts

Master algorithms control the data exchange and synchronization between Event-B and MiXiM models. Based on the master structure demonstrated in Figure 7.5, most classes are designed to be reusable. Only class `EventBCtl` has to be written from scratch for each WSN system to indicate the sequence of events in Event-B controller. This still requires the automatic script generation from this class for future work. This class's constructs can be divided into two main structures: *the algorithm steps for specific protocol operation* and *the communication and synchronisation steps between Event-B and MiXiM models* as we describe in Section 7.5. Here, we describe the implementation of algorithm steps in our master code.

The algorithm steps are represented by the sequence of events in order to perform the specific operation of communication protocols e.g. sensing and broadcasting mechanisms for SensorApp and parent selection for MintRoute. As in Event-B's semantic, the event executed next is chosen non-deterministically from the set of enabled events. Based on our surveying studies in the literature of master implementation on ProB 2.0 (ADVANCE, 2013; Savicks et al., 2014), they implement determinism at the final refinement model for translation to a master code. The event ordering is embedded into event

guards and actions. Similar to this, we diminish the non-determinism in our Event-B controller models through refinement chains by introducing more deterministic ways of choosing events for execution. This leads our final refinement model of each protocol algorithms to be deterministic.

However, during implementing our master code, we discovered that it is possible that a node controller can be executing more than one process concurrently. For example, in the SensorScope case study, several processes can perform concurrently e.g. data sensing, beacon and route broadcasting and transmission mechanisms as shown in Figure 7.10. In our C-style simulation experiment, for instance, we indicated the periodic timer for each sensor node to sense and transmit a data packet for SensorApp protocol to every 120 seconds. This mechanism is performed concurrently with beacon broadcasting mechanism for MintRoute protocol. We set the same periodic time (120 seconds) of sensing timer to periodic beacon broadcasting timer at a sink (as in Figure 7.10(a) and (b) respectively). After a beacon is broadcast for a certain period, the link quality is executed. In our experiment, we configured the link quailty to be estimated every 3 cycles of broadcasting beacon. Thus, the first estimation was set at 470 seconds (after finishing cycle 3 of beacon broadcast). After link quality estimation finishes, route broadcasting mechanism is performed. This results in the fourth cycle of beacon broadcasting mechanism and route broadcasting mechanism to be performed concurrently as demonstrated in Figure 7.10(b) and (c) respectively.

Two concurrent processes create the interleaving of events. For example, considering the fourth cycle of beacon broadcasting mechanism, we have two concurrent subprocesses: beacon broadcasting (*Bea*) and link quality estimation (*Lnk*). The interleaving of events caused by these two subprocesses can begin with the event enabling the node to start flooding beacon (*start_flooding*). Then, this execution is followed by link quality estimation due (*est_due*), the link quality calculation and update (*update_est*) respectively before performing create beacon event (*create_beacon*). The trace demonstrating interleaving of events that corresponds to the example described earlier is shown as follow.

$$\langle Bea.start\_flooding, Lnk.est\_due, Lnk.update\_est, Bea.create\_beacon, ...\rangle$$

We found that this interleaving of events causes more than one event to be enabled at the same time in order to support concurrent subprocesses. This results in non-deterministic choice in the Event-B model. In order to distinguish separately these concurrent mechanisms, we replace this non-determinism at the node level with a schedule that defines an execution order on the events in the master code. Our developed master compensates for per-node determinism that would be presented at the bottom level of refined Event-B machine. An example of sequence of events demonstrating concurrent subprocesses is shown in Figure 7.10.

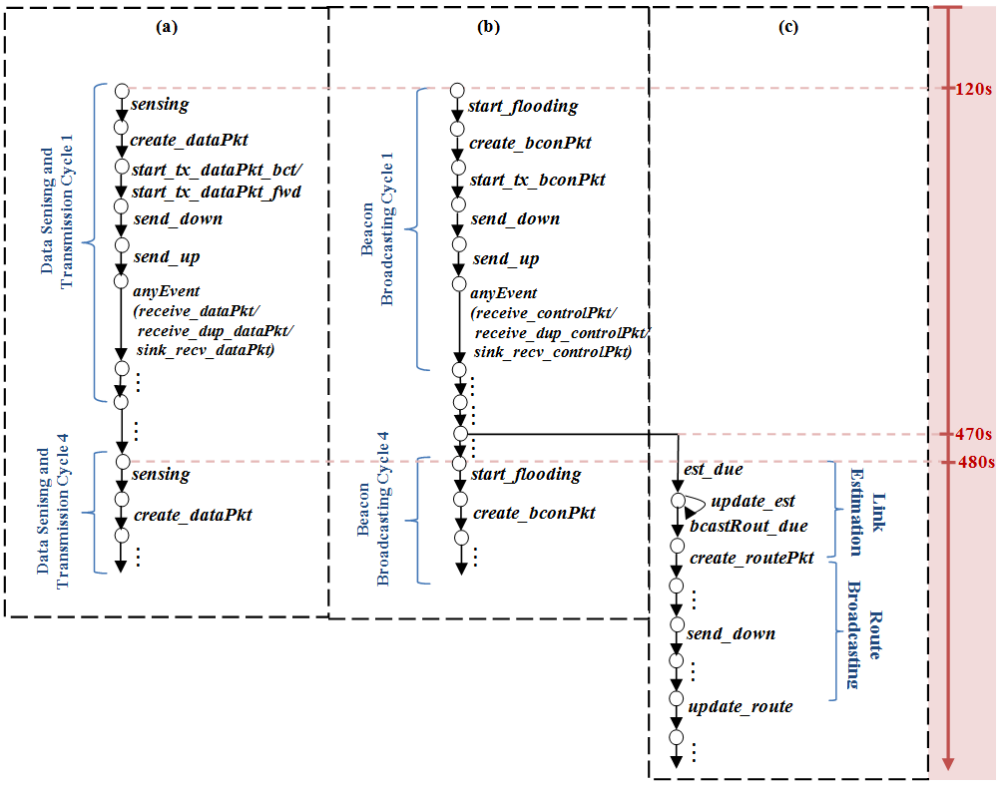


Figure 7.10: concurrent traces performing (a) data sensing and transmission, (b) beacon broadcasting and (c) link quality estimation and route update packet broadcasting mechanism

Figure 7.11 shows method `run()` implementing different concurrent mechanisms. Each concurrent mechanism is distinguished by variable `threadType` that is input as a parameter of this method. Three mechanisms mentioned above are implemented: data sensing and transmission (`threadType 0`), beacon broadcasting (`threadType 1`) and link quailty estimation and route update broadcasting (`threadType 2`) mechanisms. Method `transmitData()` (line 8, Figure 7.11) encodes the sequence of events to form up data sensing and transmission mechanisms as shown in Figure 7.10(a). Examples of code of this mechanism is explained in Figure 7.13 and 7.14, the next section. Similar to this method, methods `floodBeacon()` (line 14, Figure 7.11) and `estCost()` (line 17) encode the sequence of events performing beacon broadcasting mechanism, and link quality estimation and route broadcasting mechanism as demonstrated Figure 7.10(b) and (c) respectively.

```
1:  …
2:  public void run(int threadType,int nd){
3:      switch(threadType){
4:          case 0: //case 0 for sensing data
5:                  sensedFlg[nd] = mintRouteCtl[nd].getSensedFlg(nd,…)
6:                  sensingNDs[nd] = mintRouteCtl[nd].getSensedNode(nd)
7:                  sensedData[nd]=mintRouteCtl[nd].getSensedData(nd)
8:                  if(sensedFlg[nd]){
9:                      mintRouteCtl[nd].transmitData(sensingNDs[nd],sensedData[nd],…)
10:                 }//end of if
11:                 break;
12:         case 1: //case 1 for beacon broadcast
13:                 boolean floodBeacon= mintRouteCtl[sinkAddr].getFloodBeacon()
14:                 if(floodBeacon)
15:                   mintRouteCtl[sinkAddr].floodBeacon(startTime,tempCount)
16:                 break;
17:         case 2: //case 2 for link quality estimation and route flooding
18:                 mintRouteCtl[index].estCost(estTick,…)
19:                 break;
20:     }//end of switch
20: }//end of void run()
```

1:	Method *run()* contains parameters thead type (*threadType)* and node id (*nd*)
2:	Selecting thread type from the parameter:
3:	- Case sensing data (*threadType=0*)
4:	+ Extracting sensing information and retrieving sensed flag (*sensedFlag*)
5:	+ retrieving sensing node (*sensingNDs*)
6:	+ retrieving sensed data (*sensedData*)
7-9:	+ If there is sensed data for a particular node *nd,* Calling method *transmitData()* in object *eventCtl* (an instance of classes *EventBCtl*) to perform sensing and data transmission mechanism
11:	- Case beacon broadcast (*threadType=1*)
12:	+ Extracting beacon broadcast information and retrieving flood beacon flag (*floodBeacon*)
13-14:	+ If there is flood beacon flag for a sink (*sinkAddr*), Calling method *floodBeacon()* in object *eventCtl* (an instance of classes *EventBCtl*) to perform beacon broadcasting mechanism
16:	- Case link quality estimation and route flooding (*threadType=2*)
17:	+ Calling method *estCost()* in object *eventCtl* (an instance of classes *EventBCtl*) to perform link quality estimation mechanism and then route update packet flooding mechanism

Figure 7.11: Example of master code for distinguishing concurrent traces: data sensing and transmission, beacon broadcasting and link quality estimation and route update packet broadcasting mechanisms

7.7.2 Example of Master Algorithm

This section gives examples of the implemented master algorithm. Figure 7.12 illustrates the partial master code for initialisation phase for SensorApp protocol co-simulation corresponding to Figure 7.8. This code is defined in `MainClass` shown in Figure 7.5. This figure demonstrates the initialisation step at the master. The configuration defined by users in MiXiM is passed to the master in order to initialise Event-B controller. We implement the receiving socket program at the master in order to receive this initialisation parameter that are a number of nodes (`numNDs`) and sink id (`sinkAddr`) from a sender socket program in MiXiM's `FMInterface`. This receiving socket program is implemented in methods `receive()` of class `WSNSocket` as demonstrated in this figure. After the master receives the initialisation parameter, the multiple instances of Event-B

controller model are created, loaded and initialised. Labels `mst` and `ctl` denote the master and controller commands respectively.

```
1:  WSNSocket wsnSocket = new WSNSocket()
2:  try{
3:    wsnSocket.connect(port)
4:    String rcvMsg = wsnSocket.receive("MiXiM initialisation information");
5:    wsnSocket.shutdown(server, socket)

6:    def initInforArr = rcvMsg.split(",")
7:    numNDs = Integer.parseInt(initInforArr[0]);
8:    sinkAddr = Integer.parseInt(initInforArr[1]);

9:            for(int i=0;i<numNDs;i++){
10:                   dCtl[i] = api.eventb_load("/SensingData/CTL2.bum") as Trace
11:                   dCtl[i] = dCtl[i].anyEvent();
12:                   dCtl[i] = dCtl[i].anyEvent();
13:           }//end of for
14: }catch(UnknownHostException e){
15:     recordLog("Cannot connect to MiXiM Simulation")
16:     System.exit(1)
17: }catch(IOException e){
18:     recordLog("Failed to receive a parameter from MiXiM")
19: }//end of try
```

1:	Creating a socket object (*mst*)
3:	Establishing a connection to MiXiM (*mst*)
4:	Receiving initialisation information from MiXiM (*mst*)
5:	Shutting down the socket connection (*mst*)
6:	Extracting the received initialisation information (*mst*):
7:	- Assigning the number of nodes (*mst*)
8:	- Assigning the sink id (*mst*)
9:	For each node controller:
10:	- Creating an instance of Event-B controller model and loading it into each node controller (*ctl*).
11:	- Setting up constants of Event-B controller model in each node (*ctl*)
12:	- Initialising machine of Event-B controller model in each node (*ctl*)
14-16:	Checking whether the connection of MiXiM can be established (*mst*)
17-19:	Checking whether the initialisation information can be transmitted successfully from MiXiM (*mst*)

Figure 7.12: Partial master code for initialisation phase for SensorApp protocol co-simulation corresponding to Figure 7.9

Figure 7.13 illustrates the partial master code for sensing mechanism for SensorApp protocol co-simulation corresponding to step R1 in Figure 7.8. This code is defined in method `getSenseFlg()` of class `EventBCtl` expressed in Figure 7.5. This sensing mechanism is indicated by MiXiM simulation that transmits sensing parameter as a string stream consisting of `sesingFlg`, `sensingND` and `sensedData`. After the master receives this sensing parameter, it extracts this parameter and indicates the corresponding node to sense and record data from the environment. For example, parameter "TRUE,1,12.84" manipulated by the master to indicate node 1 to sense and record sensed data containing value 12.84. After the master extracts this parameter, method `transmitData()` defined in class `EventBCtl` is called with sending these extracted parameters. This method is responsible for recording sensed data in sense

buffer, creating and transmitting a data packet to a channel in MiXiM. This is expressed in Figure 7.14. As can be seen in this figure, shared event *sensing* and local event *create_dataPkt* is called by a master to operate sense and create a data packet in Event-B controller respectively. Method `transmit2Channel()` of the same class is called to send this created packet down to a channel in MiXiM.

```
1:  master.connectSocket(port)
2:  String senseMsg=master.receive("initial sensing information")
3:  master.shutdownSocket()

4:  def senseMsgArr = senseMsg.split(",")

5:  sensedFlg[nd] = senseMsgArr[0]=="TRUE"
6:  sensingND[nd] = Integer.parseInt(senseMsgArr[1])
7:  sensedData[nd] = Double.parseDouble(senseMsgArr[2])

8:  if(sensedFlg[nd]){
9:    eventCtl[nd].transmitData(sensingNDs[nd],sensedFlg[nd],sensedData[nd],...)
10: }//end of if
```

1:	Establishing a socket connection to MiXiM (*mst*)
2:	Receiving the sensing information from MiXiM (*mst*)
3:	Shutting down a socket connection (*mst*)
4:	Extracting the received sensing information (*mst*)
5:	Assigning sensed flag (*mst*)
6:	Assigning sensed node (*mst*)
7:	Assigning sensed data (*mst*)
8:	Checking whether there is sensed data from MiXiM environment (*mst*)
9:	If there is sensed data for a particular node *nd*, Calling method *transmitData()* in object *eventCtl* (an instance of classes *EventBCtl*) to perform sensing and data transmission mechanism

Figure 7.13: master code for sensing mechanism implemented in method `getSenseFlg()` of class `EventBCtl`

```
1:  Public void transmitData(int nd, boolean sensedFlg, double sensedData){
2:    int sensedDataInt = sensedData * 100;
3:    dCtl[nd] = dCtl[nd].sensing("x="+ nd +"&sf="+ sensedFlg +
          "&sd="+sensedDataInt)
4:    dCtl[nd] = dCtl[nd].create_dataPkt("s="+sensingNDId)
5:    transmit2Channel(nd, dCtl,...)
6:  }}//end of transmitData
```

1:	Method *transmitData()* contains parameters: sensing node ID (*nd*), sensed Flg (*sensedFlg*) and sensed data (*sensedData*) identified by MiXiM environment (*mst*)
2:	Converting *sensedData* from double to integer value (as Event-B does not support double data type) (*mst*)
3:	Indicating node controller *nd* to sense and record data (*ctl*)
4:	Creating a data packet and attaching sensed data into the created packet (*ctl*)
5:	Calling method *tranmit2Channel()* in order to transmit such a created packet to a channel in MiXiM (*mst*)

Figure 7.14: master code for sensing and creating a data packet implemented in method `transmitData()` of class `EventBCtl`

Figure 7.15 demonstrates the partial master code for packet transmission mechanism that we implement in method `transmit2Channel()` in class `EventBCtl`. This illustrates steps R2 and R4 in Figure 7.8 in which shared events *send_down* and *send_up* are used to pass the packet parameter (`chnPar`) from sender node controller down to a channel in MiXiM and return the neighbour list (`nbrLst`) from MiXiM to indicate the receiving node controller. Note that we implement method `receiveChnPar()` and `sendNbrs()` in MiXiM's `FMInterface` to receive parameter `chnPar` from and return parameter `nbrLst` to Event-B controller (as illustrated in Figure 7.7). The socket program is used to exchange these parameters between a master and MiXiM. This packet transmission operation is repeated until there are no packets recorded in the waiting buffer (*ndBuff*) of each node.

```
…
while(!(dCtl[nd].getCurrentState().value("ndBuff")=="{}")){

    dCtl[nd] = dCtl[nd].start_tx()
    extractPktField()
    dCtl[nd] = dCtl[nd].send_down("cn="+ctlCurFwdr+"&pkt="+ctlCurPkt+
               "&sno="+ctlCurSeqNo+"&dest="+ctlCurDestAddr)

    String chnPar = generateChnPar(nd, ctlCurFwdr, ctlCurPkt, ctlCurSeqNo,
                    ctlCurDestAddr,ctlCurData)

    master.connectSocket(port)
    master.send(chnPar)
    nbrLst = master.receive("neighbour list information")
    master.shutdownSocket()

    nbrs = extractNbrLst(nbrLst)

    for(int nbr =0; nbr < nbrs.length; nbr++){
        dCtl[nbr] = dCtl[nbr].send_up("nb="+nbrs[nbr]+"&pkt="+envFPkt)
        dCtl[nbr] = dCtl[nbr].anyEvent()
    }//end of for
}//end of while
```

1:	Checking whether there are packets in a waiting buffer of each node being transmitted (*mst*)
2:	Selecting and retrieving a packet recorded in a waiting buffer of node *nd* (*ctl*)
3:	Extracting packet fields to prepare input parameter
4:	Indicating sending packet down operation in each node controller (*ctl*)
5:	Generating the packet information parameter (*chnPar*) (*mst*)
6:	Establishing a socket connection to MiXiM (*mst*)
7:	Sending parameter *chnPar* to MiXiM (*mst*)
8:	Receiving the neighbour list (*nbrLst*) back from MiXiM (*mst*)
9:	Shutting down a socket connection (*mst*)
10:	Extracting each neighbour in the received neighbour list (*mst*)
11:	For each neighbour node:
12:	- indicating sending a transmitted up to each neighbour (*ctl*)
13:	- each neighbour node (including a sink) receives a transmitted packet (*ctl*)

Figure 7.15: master code for packet transmission mechanism implemented in method `transmit2Channel()` of class `EventBCtl`

7.8 Co-simulation Validation

In order to validate our C-style prototype, we ran our C-style co-models within 1400 seconds of simulation time with different network topologies varying from 1-hop to 3-hop network containing 4 to 7 nodes respectively. These simulation networks were generated from the parameters configured in the MiXiM configuration file. This performs eleven cycles of sensing periods, together with three cycles of parent selection. There are two comparison criteria considered to validate this C-style prototype. Firstly, we compare the event trace of F-style co-simulation to that of C-style co-simulation. We expect that the event trace generated in both style development must create the same input parameter (`chnPar`) and output parameter (`nbrList`). Secondly, we compare the performance analysis evaluated by S-style simulation to that evaluation by C-style co-simulation. The result of this comparison is expected that the same pattern of performance analysis must be conducted by these two styles.

7.8.1 Comparison of Event Trace in F-style and C-style Co-simulation

Figure 7.16 shows an example of master execution results of C-style co-simulation compared to F-style co-simulation at application and network layers. It can be seen that they create the same execution results. For SensorApp protocol, Figure 7.16(a) shows the partial execution result created by F-style models. This illustrates broadcasting packet mechanism by node 2 in which packet PKT11 created by node 1 is broadcast to node 2's neighbours. These neighbours are specified by the environment model and return them as a list to the controller. After the transmission, packet PKT11 is received by a sink and detected as a duplicated packet by nodes 1 (as its own packet), 3 and 4 (as received it already). The co-simulation results in Figure 7.16(b) are the same as in Figure 7.16(a). To connect to MiXiM environment model, the connection for the sockets at both sides is established for exchanging channel information and neighbour list parameters. Figure 7.16(c-d) shows the same route tree created by pure Event-B and co-simulation models for MintRoute protocol. This route tree corresponds to one possible route tree created by S-style models described in Chapter 5.

Furthermore, based on this development result shown in Figure 7.16, we discover that the method trace of the MiXiM environment creates the concrete version of the event trace of the Event-B environment which provides the same output. From this result, for example, the single event trace of Event-B environment model which reported the neighbour list of forwarder node 2 is

$$\langle find\_neighbours.2.\{0, 1, 3, 4\}\rangle.$$

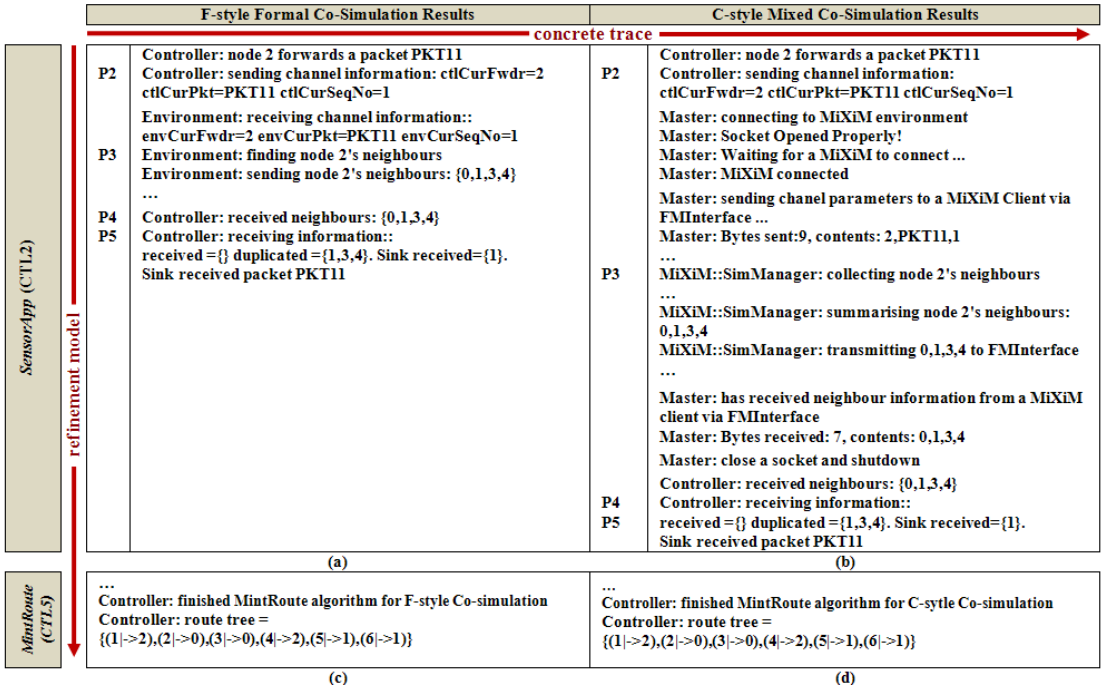


Figure 7.16: Master algorithm for SensorApp protocol co-simulation

The corresponding MiXiM trace is as follows (note that `rptRecv`.x.y is the MiXiM environment method using wireless connectivity implicitly to find each neighbour node y of forwarder x in the network):

$\langle$`rptRecv`.2.0, `rptRecv`.2.1, `rptRecv`.2.4, `rptRecv`.2.3, `sendNbrs`.2.{0,1,3,4}$\rangle$

If we remove the method `rptRecv` for this above trace, we will achieve the trace of MiXiM environment as $\langle$`sendNbrs`.2.{0, 1, 3, 4}$\rangle$. We see that the F-style communication trace is a subtrace of the C-style trace.

7.8.2 Comparison of Performance Analysis in S-style Simulation and C-style Co-simulation

We compared the performance analysis results in C-style with that of S-style. We analyse the qualitative comparison in order to identify whether the analysed profiles between S- and C-style are similar or different. We expected that these analysed profiles are comparable.

Figure 7.17, for example, shows one of our co-simulation results from the network containing the topology illustrated in Figure 5.4. As shown in Figure 7.17(a) for both styles, the latency of node 2 is quite similar. This latency is always very low as node 2 always chooses a sink to be a parent. However, we discovered that the randomness of the routing algorithm to discover the route tree affects this performance analysis. In Figure 7.17(b), at two-thirds of running simulation time, the latency of node 5 in S-style

simulation is considerably higher. This is because its parent had the highest number of forwarded data packets compared to that of C-style.

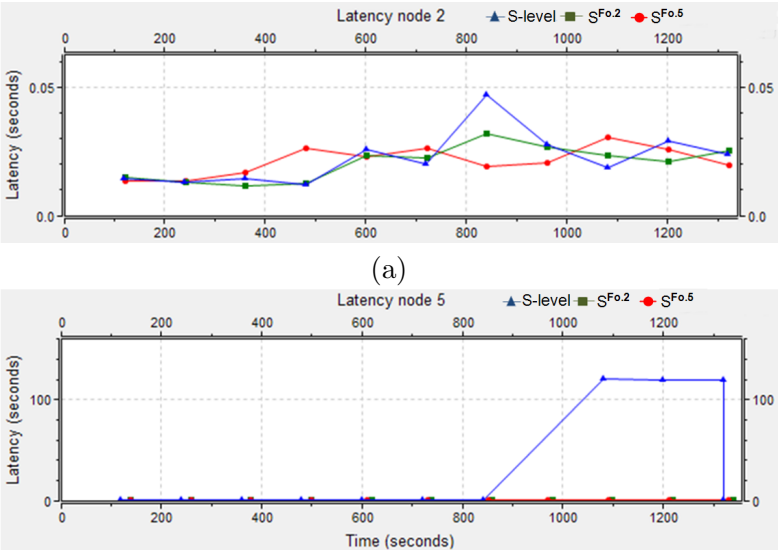


Figure 7.17: Example of performance analysis results of some nodes

7.9 Flexibility of Co-simulation Development

In the previous section, we have demonstrated that our FoCoSim-WSN framework enables the integration methodologies for WSN development: (S)imulation, (F)ormal and (C)osimulation. During development, this framework provides an iterative interworking scheme between the formal node controller and the physical simulation environment through multiple refinement levels. This framework provides options for system engineers, in which they can implement their models by cross over into these three flexible modes of working easily. This section describes the utility and flexibility of FoCoSim-WSN framework that is classified into two styles of interworking evolution. One is *from-S-through-F-to-C* (S^F *Co-simulation*) in which the development of co-simulation starts from S-style through F-style development. The other is *from-F-through-S-to-C* (F^S *Co-simulation*) in which the development of co-simulation starts from F-style through S-style development. Firstly, we explains a fundamental co-simulation development approach for these two styles of interworking evolution in Section 7.9.1 before S^F *Co-simulation* and F^S *Co-simulation* are described in Section 7.9.2 and 7.9.3 respectively.

7.9.1 Fundamental Approach for Co-simulation Development

We have discovered that MiXiM development in S-style enables the style of bottom-up development approach in which the WSN system development starts with existing

primitive constructs provided by each protocol layer are integrated from the bottom represented by the physical layer to the top represented by the application layer. Then, each primitive layer is constructed gradually to create more and more complex features. This is accomplished by adding the functionality of specific protocols on the top, until all the protocol algorithms have been created (as demonstrated in Chapter 5). As demonstrated in Figure 7.18(a), for example, each layer of protocol stack provides base modules (general structure) that contain only packet construction/encapsulation and decapsulation for data transmission and standard gate interface for sending/receiving packet between each layer. We extended these modules by adding specific functionality to implement specific protocol algorithms (e.g. SensorApp, MintRoute and S-MAC) on the top. Then, these implemented protocol algorithms together with simple physical model (*BasePhy*) are integrated to form up the protocol stack for WSN system.

On the other hand, in F-style development, Event-B facilitates the top-down refinement-based development approach. Our protocol algorithm is gradually development by encoding the model from simple to complex requirements through refinement steps. Finally, for each final model of each protocol algorithm, shared-event decomposition technique in Event-B enables the verified software controller model that will be generated to node code, to be separated from the environment model as we demonstrate in Chapter 6.

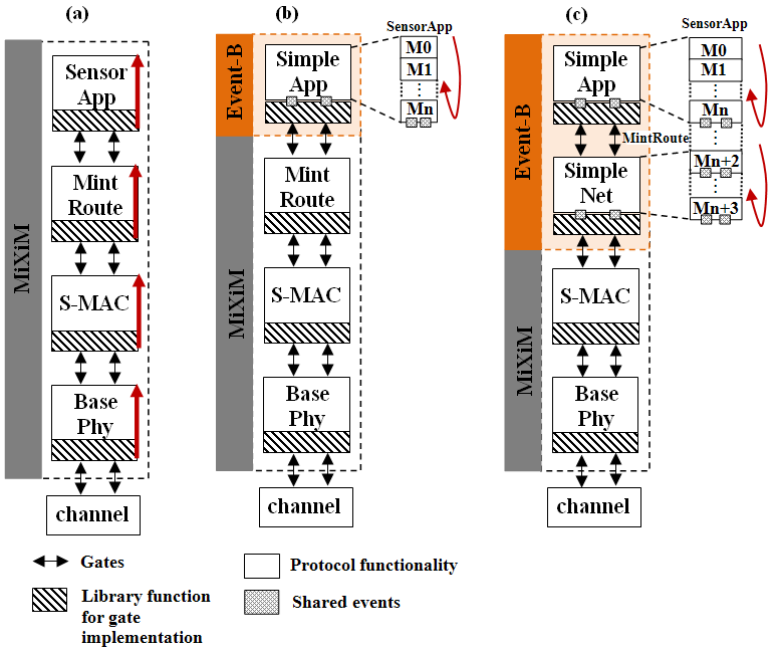


Figure 7.18: The fundamental Approach for Co-simulation Development combing (a) the bottom-up MiXiM approach and (b-c) the top-down Event-B approach for application and network protocols respectively

Our C-style development combines the top-down approach from F-style development to implement controller models and the bottom-up approach from S-style development to integrate formal controller models in Event-B with environment models in MiXiM.

In MiXiM, we retain only physical and environment models. The bottom-up approach prepares the primitive protocol stack that integrates the lower layer protocols like MAC and physical models with the upper layer protocols like application and routing models. Our FoCoSim-WSN framework provides SimpleApp and SimpleNet as primitive application and network layers that co-ordinate and communicate with the formal controller models of upper layer protocols as demonstrated in the earlier section. The top-down development approach enables each upper protocol to be developed gradually through refinement steps as shown in Figure 7.18(b) and (c) for SensorApp and MintRoute protocols as examples respectively. This fundamental approach for co-simulation development provides two styles of interworking evolution: S^F *Co-simulation* and F^S *Co-simulation* that are described in the following sections.

7.9.2 S^F Co-simulation Development

This development technique is for the legacy or existing MiXiM models which need formalizing into Event-B. This development technique starts with the S-style development which already contains the existing protocol algorithm layered in a protocol stack in MiXiM framework. Then, the controller algorithm implemented in this pure simulation model can be separated from the environment model and implemented in Event-B as we have already demonstrated in Section 7.6. The separated environment still retains only the communication gates (no algorithm) together with the standard function in MiXiM such as packet en/decapsulation as described in Section 7.4.3. For two upper layers, we developed standard modules `SimpleApp` and `SimpleNet` for application and network layers respectively. These two standard modules can be reused for the potential development which enables the formal controller model for these two upper layers to communicate with other lower layers of physical environment in MiXiM. Another example of this development mode is our implemented S-MAC models in MiXiM as shown in Figure 7.19. This will separate the controller out and retain the standard interface for MAC layer in MiXiM. The S-MAC algorithm controller in Event-B can communicate to the adjacent layer by using this interface. This is still work in progress.

For the master implementation, we have designed our master structure to be reusable for future development as shown in Figure 7.5. This enables the generic class including main class (`MainClass`), master class (`Master`) and multithread creation (`Task`), socket program (`WSNSocket`) and log file management (`LogMgt`) to be totally reused. Class `EventBCtl` that creates and drives the sequence of the event (the event trace) is required to be adjusted in order to access the state of variables in the developed Event-B controller e.g. for S-MAC in this example. However, based on the SensorApp and MintRoute protocol development, we have discovered that these two layer protocol models share the packet transmission mechanism as a generic process. The packet transmission is required by any protocol algorithm to determine the specific process e.g.

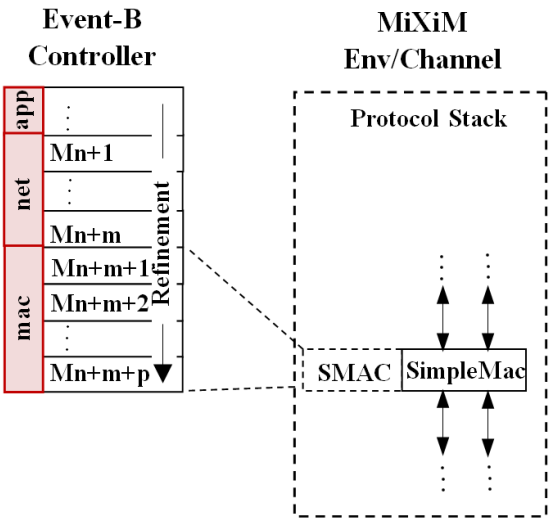


Figure 7.19: Example of $S^F Co - simulation$ development of our S-MAC protocol

updating neighbour table and estimating link quality operations in MintRoute protocol after receiving nodes receive a transmitted beacon packet. This generic process creates the generic co-simulation pattern which performs repeatedly to transmit any packets e.g. data and beacon packets in waiting buffer (*ndbuff*) until there are no packets in the buffer (this is encoded into method `transmit2Channel()` in Figure 7.5 and 7.15). Therefore, class `EventBCtl` for potential system development can be implemented by reusing and extending this packet transmission pattern and only changing the sequence of events enabling specific protocol operation.

7.9.3 F^{S}Co-simulation Development

This development technique provides a strategy which layers the protocol algorithm for formal verification. It starts with F-style development in which the protocol algorithm is layered through refinement steps in Event-B before it is integrated with the environment models in MiXiM. In MiXiM environment, system engineer can extend the base layer (*BaseLayer*) provided by MiXiM for each layer of protocol stack in order to connect to the formal controller as demonstrated in Figure 7.6. However, instead of developing this from scratch, reusing our standard module for communicating between two upper layers (such as *SimpleApp* and *SimpleNet*) is another way that can be used to prepare simulation environment as shown in Figure 7.20.

For the master implementation, the generic classes defined in Figure 7.5 can be totally reused to develop a master for the new development. This includes classes `MainClass`, `Master`, `Task`, `WSNSocket` and `LogMgt`. As the new development may contain different protocol algorithms, class `EventBCtl` is needed to be entirely re-developed. However, the generic packet transmission mechanism expressed in the previous section can

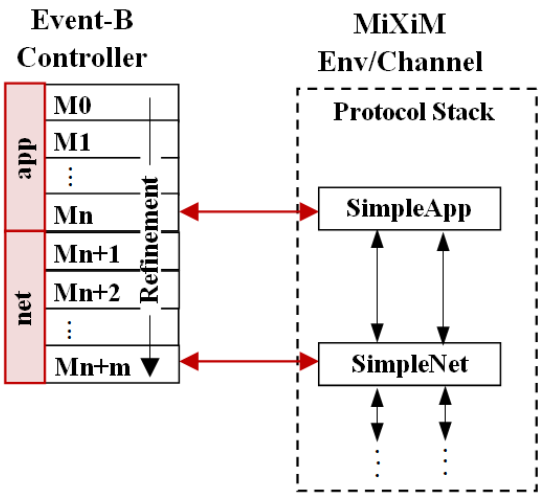


Figure 7.20: Example of $F^S Co - simulation$ development of our S-MAC protocol

be reused in this class development. To improve the master structure, we implement base class `EventBCtl` to contain standard methods for sensing mechanism (methods `getSensedFlg()`, `getSensedData()`) and packet transmission mechanisms (methods `transmit2Channel()`) as shown in Figure 7.21. This class can be extended to implement the specific class in which these generic mechanisms are derived and the specific operation can be added into this implemented class. In Chapter 9, we describe this F^S *Co-simulation* development with its reusability in detail. It is demonstrated with the different case study, Road Tunnel Monitoring and Control System (RTMCS).

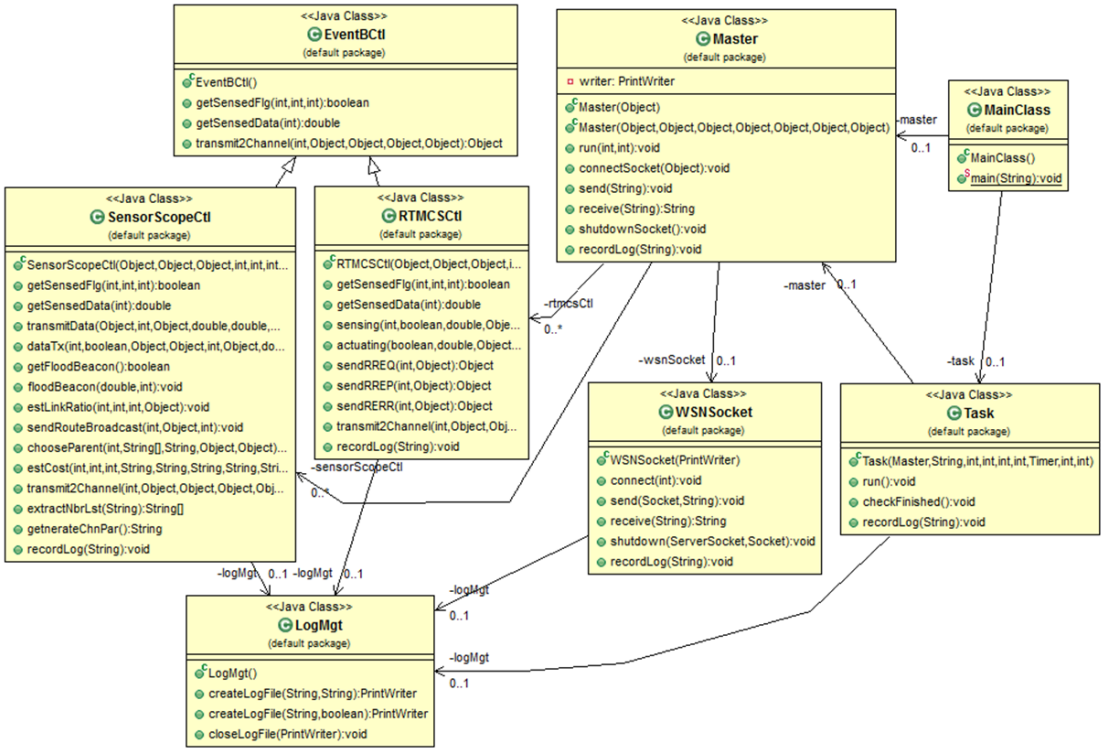


Figure 7.21: Reusable master structure

7.10 Co-simulation Implementation Effort

In earlier sections, we present and explain the co-simulation framework architecture with its implementation. To better understand this co-simulation development process and product, we characterise and measure the framework components together with their implementation effort. This section provides details of the implementation effort required to develop a co-simulation framework that can possibly be used as a guideline to predict the amount of effort required for further development or maintenance of co-simulation framework.

In our co-simulation framework implementation, two co-simulation platforms are implemented that are Master and MiXiM extension. Based on our actual time of the framework implementation, we can measure that the implementation of our co-simulation framework requires approximately 4.5 man-months of the total implementation effort. Of this figure, 2 man-months are required by the implementation of extended MiXiM platform. This is calculated from the effort of coding extended interface and the effort required for studying on the MiXiM structure and the use of standard libraries provided by this simulator - implementing sending/receiving packet, packet encapsulation/decapsulation and socket interface. The implemented MiXiM extension contains 4 classes including FMInterface, SimManager, SimpleApp and SimpleNet, 71 methods and 2,809 lines of code in total. Since these extended MiXiM interfaces are designed to be reusable

components, we believe that most classes in this structure can be totally reused for the potential development.

On the other hand, we have discovered that 2.5 man-months are required by the master implementation which is slightly greater than the effort required by the extended MiXiM implementation. This is for two reasons. Firstly, it requires the time effort to design and implement the master algorithm in order to drive the sequence of events to form up the specific mechanism of communication protocols e.g. link quality estimation and parent selection for the MintRoute. The second reason is that the multi-thread and scheduling mechanism are required to be implemented in this master platform in order to generate multi-threads of node controllers and schedule them to repeat the mechanism periodically. Thus, this requires the effort to implement these mechanisms. Furthermore, this implementation requires not only the effort of coding master but also the effort required for studying the use of standard libraries including socket program interfaces and APIs provided by ProB 2.0 for accessing the states and events in Event-B machine. This results in 6 classes, 67 methods and the total line of code (LOC) is 1,538 to be implemented in our master platform. Similar to extended MiXiM components, we designed our master structure to be a reusable platform. Most classes in this structure can be totally reused for the potential development. The reuse of master and MiXiM components is demonstrated in Sections 7.9 in this chapter and Section 9.5 in Chapter 9.

Since the integration and testing affected the change of code in our co-simulation framework, the effort of this integration and testing is also measured as one part of implementation effort. We integrate the implemented master and extended MiXiM platforms and test the communication and parameter exchange between them by co-modelling with two case studies. This first case study is a simple case study in order to enable the primary platform integration and communication testing. Another case study is more complex, the SensorScope, in which we have explained the detail of this validation and testing in Section 7.6. Here, we describe the platform integration and communication testing by using the simple case study.

The simple case study contains the SensorApp that is used in our SensorScope project for application protocol. For routing protocol, we implement the SimpleRoute protocol that consists of two steps, *neighbourhood discovery* and *route tree construction*. The neighbourhood discovery step in this routing protocol is similar to that of MintRoute protocol in which it begins with broadcasting a control packet initiated by a sink to its neighbours. When a sink's neighbour receives the broadcast control packet, it re-broadcasts this packet to its neighbours and so forth. For the route tree construction, the source of the first received flooded control packet is selected to be a parent of the receiver. For testing steps, we ran our co-simulation with 1 sensing cycle (120 seconds) in order to validate and test the communication steps between the master and MiXiM

environment. Then, we ran our co-simulation with longer simulation time (1400 seconds) in order to validate and test the performance calculation and measurement. We also ran the experiment with a more complex case study, the SensorScope, to validate and test the integration of our developed framework as demonstrated in Section 7.6. At the end of the experiment and bug correction from these two case studies, we can conclude that 2 man-months (1 man-month for each case study) are approximate total efforts required by this co-modelling for platform integration and testing.

7.11 Conclusion

This chapter has demonstrated the co-simulation approach to the extension of current SE for WSN development. Our prototype development shows that the framework integration for (F)ormal, (S)imulation and (C)osimulation can combine the benefits between formal and simulation based modelling approaches. Our FoCoSim-WSN framework provides high-level abstraction and separation of concerns for WSN development that reduces the complexity of development. By using this framework, the node controller encoding protocol algorithms is modelled separately from hardware level models e.g. analogue and radio models, and the physical environment where the nodes are deployed. Co-simulation is an approach for the joint simulation of models developed with different tools. Furthermore, the framework also enables strong V&V which co-simulates between the verified node controller model in Event-B and stochastic environment model in MiXiM. This confirms the sub-contribution C.1 that is the integration between proof-based verification and simulation-based testing techniques.

The result of C-style development demonstrated in this chapter has shown that our prototype *FoCoSim-WSN* provides multiple abstraction levels for co-simulation through refinement. We have demonstrated the iterative and incremental development in SensorScope case study which enables three levels of abstraction: no partition of development (*S-level*), the abstraction level of application protocol ($S^{Fo.2}$ for SensorApp) and the abstraction level of network protocol ($S^{Fo.5}$ for MintRoute) as shown in Section 7.6. Furthermore, we have discovered two styles of interworking evolution: S^F *Co-simulation* and F^S *Co-simulation*. System engineers can also work either F- or S-style development before the separated controller and environment are combined and co-simulated in our prototype framework. This framework is flexible and can be utilized to integrate between F-, S- and C-style modelling. The independent interfaces implemented in this framework enable flexible and sufficient co-simulation communication between the formal Event-B controller and MiXiM simulation environment models through exchanging standard XML input/output parameters. System engineers can implement their models by moving between these three modes of working. This confirms sub-contribution C.2 and C.3.

Note that this chapter aims to demonstrate S^F *Co-simulation* development that starts with the existing protocol model in S-style to C-style through F-style development. The F^SCo-simulation development together with the reusability of FoCoSim-WSN framework is demonstrated with the different case study is expressed in Chapter 9. Furthermore, the next chapter demonstrates the application of model-based trace testing (MBTT) technique to strengthen V&V technique in our formal co-simulation framework. We have discovered that this technique benefits the validation coverage in the formal Event-B controller in co-models.

Chapter 8

MBTT Approach for Validation of Formal Co-simulation Models

8.1 Introduction

In previous chapters, we proposed a Formal Co-Simulation (FoCoSim-WSN) framework for WSNs confirming that our verified target formal controller models satisfy their specification including functional, safety and non-functional properties under different long running environmental conditions in a simulation engine. However, in terms of functional validation, this framework only provides the coarse-gained testing in which test scenarios are generated randomly in a simulation environment to validate the controller model. These will be many unexplored scenarios that might reveal flaws e.g. incorrect constraints and events in the executable formal model. Consequently, a fine-gained testing approach supporting validation coverage by using a trace testing technique is introduced in this work.

This chapter demonstrates controller model validation in WSN co-simulation by applying a model-based trace testing technique. This technique can derive test scenarios for functional, failure and recovery testing from a formal Event- B model, the model under test (MUT). Different testing experiments are created from these test scenarios and configured in the configuration file for MiXiM, test driver in the co-simulation framework. This work shows how "failure test cases" ("killer traces") containing failure scenarios can influence the model debugging to reveal the absent or erroneous constraints and events.

The next section presents a model-based trace testing (MBTT) approach for Formal Co-Simulation. Section 8.3 describes the validation mechanism influenced by the MBTT process, demonstrated using a case study. This section also describes how to specify killer traces for failure testing that enables the hidden incorrect constraints and events

to be detected and debugged. Finally, Section 8.4 discusses the results before describing the conclusion in Section 8.5.

8.2 MBTT Approach for Formal Co-Simulation

Figure 8.1 shows the Model-Based Trace Testing (MBTT) Approach for Formal Co-Simulation. This approach is a specification-based testing integrated from model-based testing (Utting and Legeard, 2007) and trace checking (Howard et al., 2011) approaches. It aims to identify test traces for the controller model testing and validation.

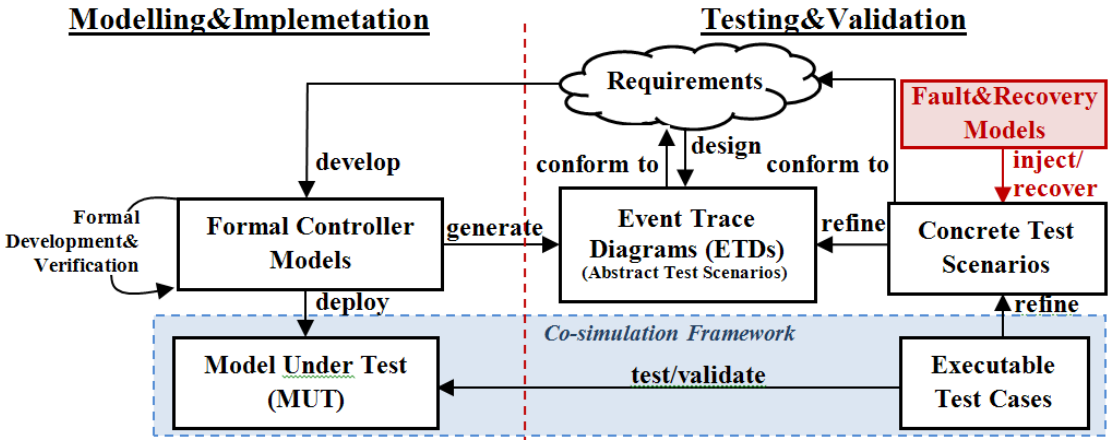


Figure 8.1: Model-based Trace Testing Approach for Formal Co-Simulation

The formal controller model, which encodes the WSN software requirements including the functionality and safety properties, is created and verified through stepwise refinement. This results in the verified formal specification of the node controller containing layered communication protocols and algorithms. This specification model is then deployed as a model under test (MUT), driven with an environment model, a test driver in the co-simulation framework for the validation process.

For testing and validation, a test scenario is one possible valid execution path enabling the MUT to be tested and validated. To accomplish this, we start with creating abstract test scenarios. We design our abstract scenarios from user requirements to represent a functional behaviour of the system. Then, we define the detail of these testing scenarios as a finite sequence of events which is derived from the formal controller model. Event trace diagrams (ETDs) are used to describe these abstract test scenarios. These diagrams express the sequence of events in the Event-B model representing the scenario of the controller behaviour. In our formal development, the achieved formal controller is a non-executable model in which a number of events are enabled to create the non-deterministic choices of occurrences. It requires to be scheduled the occurrence of events to perform the specification operation such as packet transmission. In order to drive this non-executable model in our co-simulation environment, we developed class `EventBCtl`

in the master. This class contains the scheduling task operating with the sequence of events as mentioned in Chapter 7. Consequently, instead of generating ETDs directly from this non-executable model, we generate these ETDs automatically from this master class. Furthermore, this generated ETD, viewed as an abstract test scenario, can be refined to concrete test scenarios by adding the interaction between the formal controller model and simulation environment. Fault injection and recovery are also considered in this step in order to create failure and recovery test cases. Each concrete testing scenario can be considered as an abstract representation of an executable test case. Thus, the executable test case can be derived by adding the specific parameters that will reveal errors in the execution.

8.3 MBTT in Action

This section demonstrates the action taken in applying the MBTT approach to our case study's controller model. Thus, Section 8.3.1 describes the controller modelling in Event-B in which two upper layer protocols, SensorApp and MintRoute, are layered through refinement steps. Sections 8.3.2, 8.3.3 and 8.3.4 describe how we can analyse and formalise the event trace in the controller model in order to create abstract and concrete test scenarios, respectively. Test case production and the validation results are discussed in Sections 8.3.5 and 8.3.6. "Killer traces" for detecting and debugging the hidden erroneous constraints and events are highlighted in Section 8.3.6.

8.3.1 Event-B Controller Models

As described in the previous chapter, two upper layer protocols were modelled in Event-B. The first is SensorApp for a periodic sensor application in which the sensed packet is transmitted periodically down to the lower layer. The second is MintRoute, a link quality protocol that builds the route tree from every node towards a sink. The route tree is dynamically changed based on the link quality (the successful rate of transmitted packet delivery) between nodes. The Event-B development for these two upper protocols constitutes initial model *M0* and five refinement models *M2* -*M5* as shown in Figure 8.2. In this chapter, we have interested to validate final controller model *CTL5* (decomposed model of *M5*, as demonstrated in Figure 7.9(c), Section 7.6) as it not only contains the concrete model of MintRoute protocol but also refines the final model of SensorApp protocol.

SensorApp protocol modelling: an initial model *M0* develops SensorApp as a simple abstraction over the network protocol. Then, the first two refinements *M1* and *M2* fulfil the operation for SensorApp. Shared events are implemented as a port to interact with simulation environment including broadcast packet forwarding by neighbour nodes, sensing mechanism and adjacent layer gate communication.

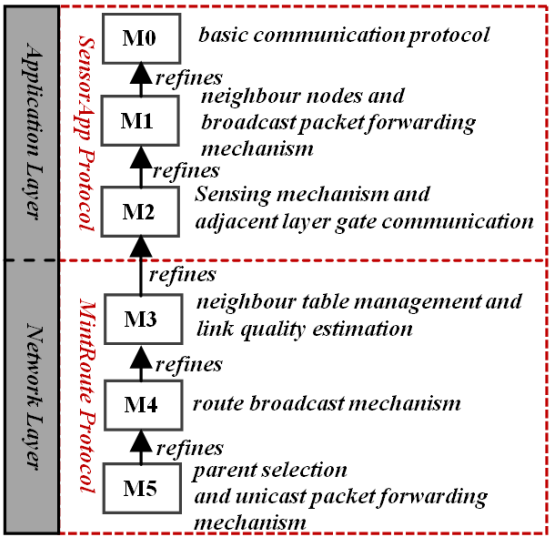


Figure 8.2: Event-B Controller Modelling Structure

MintRoute protocol modelling: the final model for SensorApp protocol (*M2*) is refined to implement the MintRoute protocol, performing four main mechanisms: neighbourhood discovery, link quality estimation, route broadcast, and parent selection. We start by creating the model *M3* by encoding neighbourhood discovery and link quality estimation before developing route broadcast mechanism in model *M4*. Finally, the final refinement model (*M5*) implements the parent selection mechanism and unicast packet forwarding mechanism via route.

Failure Detection Implementation: MintRoute provides a poor/dead neighbour detection mechanism in which each neighbour in the neighbour table has a liveliness entry. This liveliness is set at a high value when the new neighbour node is discovered. This value decreases every time when the sensor node estimates the link quality of its neighbours and is reset back when it receives a route update packet from the neighbour. Dead neighbours are identified in the neighbour table when their liveliness value becomes zero. Furthermore, this node is excluded during performing the parent selection mechanism. We implement dead neighbour detection mechanism in model *M3* and the liveliness maintenance mechanism in model *M4*.

8.3.2 Designing Abstract Test Scenarios

In our approach, we create our test scenarios starting with designing an abstract test scenario from the requirements described in Section 4.3. This testing scenario is viewed as the high level of test requirement that represents one of expected functionalities of the system. For example, in SensorScope case study, data sensing and transmission, and neighbourhood discovery is one of SensorApp and MintRoute functionality respectively. The designed abstract test scenarios for functional testing are shown in Table 8.1.

Table 8.1: Abstract test scenarios for functional testing

TS#	TS description	Requirement ID	Expected Result
TS-11	Validating sensor nodes' and a sink's functionality*	**FUN-1, FUN-2**	The sensor node should be able to sense data. Such a sensed data is transmitted from source node to a sink successfully.
TS-12	Validating broadcasting mechanism*	**FUN-3a**	A sink should be able to create a beacon packet. Such a packet is broadcast to the neighbour node successfully.
TS-13	Validating neighbourhood discovery**	**FUN-3a**	After the receiving node receives a transmitted beacon, it should record the source of transmission as a new entry of its neighbour table. The corresponding entries of the recorded neighbour e.g. the number of missed and received packet should be calculated and recorded. See Section 4.5.1.
TS-14	Validating link quality estimation mechanism**	**FUN-3b**	Each node should calculate the number of received beacons from each neighbour in indicated time interval. See Section 4.5.2.
TS-15	Validating route broadcast mechanism**	**FUN-3c**	Once the receiving node receives a broadcast route update packet, it should update the corresponding entry with the value attached in the packet. Especially, the *receiveEst* should be considered to be transmission link quality and updated into field *sentEst* in the neighbour table. See Section 4.5.3.
TS-16	Validating parent selection mechanism**	**FUN-3d**	Each node should choose its parent from the neighbour node which has the high link quality towards a sink. See Section 4.5.4.
TS-17	Validating data packet forwarding mechanism**	**FUN-3e**	Once the route tree is established, this specific neighbour node is identified from the route tree as a next node to receive a forwarded data packet. See Section 4.5.5.

* -*SensorApp* and ** - *MintRoute*

8.3.3 Event-B Model Trace Analysis

After achieving the designed abstract test scenarios in the previous section, the detail of these scenarios that enables the valid execution path to test and validate the MUT is considered. We start with analysing the event trace representing the controller behaviour. This can be expressed by an event trace diagram (ETD)(Rumbaugh et al., 1991). Firstly, we extract the system scenarios represented in terms of the finite sequence of occurring events in the interface class `EventBCtl`. This interface class encodes the final refinement of controller model *CTL5* that contains the concrete model of both communication protocols, SensorApp and MintRoute. Then, ETDs are generated automatically from this interface class by an eclipse based UML creator and generator, ModelGoon[1]. Five ETDs corresponding to SensorApp and MintRoute operations are generated. These include the data sensing and transmission mechanisms of SensorApp, beacon broadcasting, link quality estimation, route broadcasting and parent selection of MintRoute protocols. These generated ETDs also correspond to the designed abstract test scenario achieved in Section 8.3.2. They can be viewed as a detailed abstract test scenario that we use to create the concrete one described in the next section.

[1]see ModelGoon - http://www.modelgoon.org/

Figure 8.3 shows an example of generated ETDs describing the data sensing and transmission mechanisms for the SensorApp protocol. This ETD is generated from the master code demonstrated in Figure 8.3(a) that corresponds to Figure 7.12-7.15.

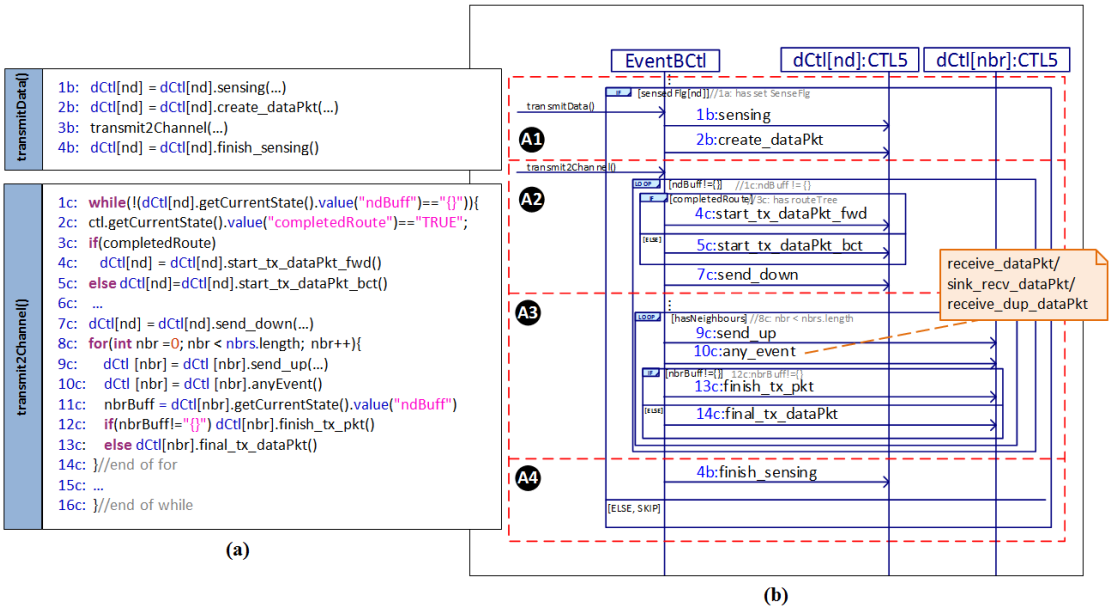


Figure 8.3: Example of abstract test scenarios transformation from (a) master code describing the trace of the data sensing and transmission to (b) ETDs representing the trace of the data

These mechanisms can be described as follows. (A1) Only active sensor node *nd* (*sensedFlg[nd]==TRUE*) senses data periodically from the environment (shared event *sensing*). Then, a data packet containing sensed data is created (event *create_dataPkt*). (A2) demonstrates the data packet transmission/forwarding process from a sender/fowarder. This starts with considering if the node has a parent (*completedRoute==TRUE*), this parent will be specified as the next node to transmit such a created packet (event *start_tx_dataPkt_fwd*) to. Otherwise, it will assign the broadcast code (-1) for the next node destination to broadcast the data packet to all neighbours (event *start_tx_dataPkt_bct*). Then, each created/forwarded packets recorded in each sender/forwarder nodes waiting buffer (*ndBuff*) is sent down (shared event *sent_down*) to the lower layer of the protocol stack and finally to the wireless channel. (A3) represents the receiving process at the receiving node, *nbr*. The transmitted packet in the wireless channel is sent up (shared event *sent_up*) to the upper layer of the receiving node. This creates the alternative choices of the receiving packet mechanism in which the receiving node including a data sink will accept the transmitted packet (event *receive_dataPkt* or *sink_recv_dataPkt*) or detect the incoming packet to be a duplicated packet (event *receive_dup_dataPkt*). Next, the completion of packet transmission is indicated (by either event *finish_tx_pkt* or *final_tx_pkt*). This transmission process (A2-A3) is repeated for forwarding a received packet if the receiving nodes are a non-destination node. Finally, (A4) initiates the finish of sensing mechanism (by event *finish_sensing*).

8.3.4 Deriving Concrete Test Scenarios

As described in Section 8.2, two types of testing are considered in MBTT approach for co-simulation, functional and failure testing. The former type concerns testing with a valid trace that checks whether our co-simulation confirms the normal behaviour of system. We derive five abstract test scenarios as ETDs achieved from Section 8.3.3 to manually identify the concrete test scenarios for the normal transmission and calculation (grouped into *TS-1*). These test scenarios fulfil information about the operation of Event-B controller and its environment implemented by MiXiM simulation. However, the automatic generation for this step is for future work.

Failure testing involves forcing our co-simulation to behave consistently with fault scenarios. We apply the fault injection technique to achieve this testing goal. Failure model has to be implemented in MiXiM in order to do the failure traces in co-simulation environment. This includes the common failures in WSN systems after the deployment, node and link failures, together with recovery mechanisms. Various failure and recovery scenarios are configured in the simulation configuration file. These are used to check whether our formal controller model can handle the occurrence of faults. To develop this, we extend each normal testing scenario described above by injecting the permanent failure scenarios which can be grouped into four categories in *TS-2* to *-5* and failure recovery as in *TS-6*. Here, the group of achieved testing scenarios can be shown as follows:

TS-1: Normal transmission including packet unicasting and broadcasting for both protocols and calculation for MintRoute protocol including link quality estimation and parent selection.

TS-2: Partial link failures e.g. in Figure 8.4(a), some links surrounding transmitting node (3) lose the connection.

TS-3: All link failures e.g. in Figure 8.4(b), all links surrounding transmitting node (3) lose the connection.

TS-4: Single dead node e.g. in Figure 8.4(c), sensing node 3 dies and it does not sense and transmit any packets. Furthermore, this dead node will not receive the broadcast packet from its neighbours such as node 2.

TS-5: Multiple dead nodes e.g. in Figure 8.4(d), the protocol algorithm should exclude dead nodes 1 and 3 during performing the specific operation such as route construction. This scenario mainly aims to evaluate the fault tolerance and recovery by injecting the massive failure.

TS-6: Link and node failure recovery which validates the protocol algorithm whether it is back to perform normally after the repair point. (This is replicated from the failure detection and repair scenario of SensorScope project)

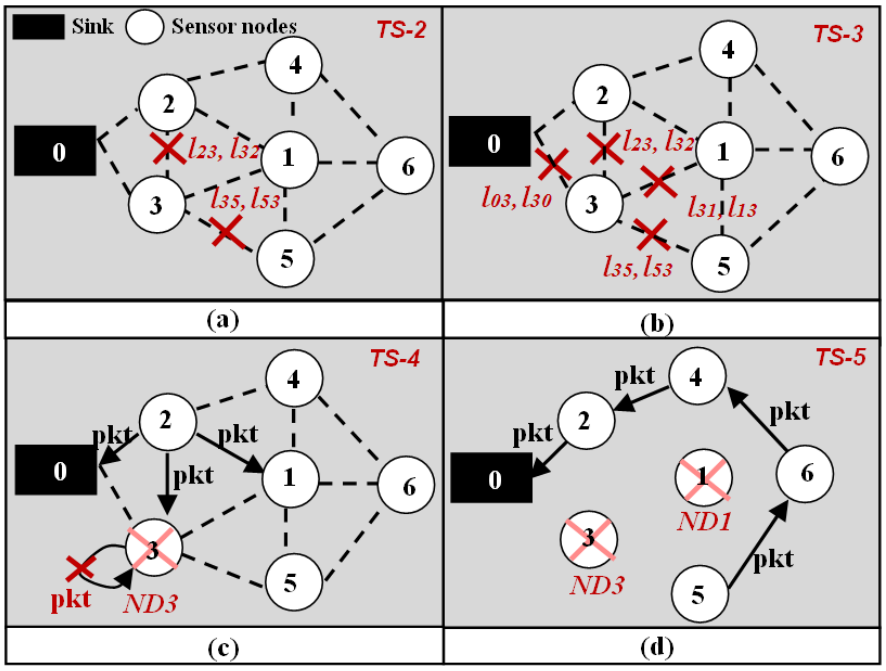


Figure 8.4: Different testing scenarios derived from ETDs

In order to generate the executable test cases, we refine our abstract ETDs to be the concrete version that interacts with the MiXiM environment model. Since failure model is implemented in MiXiM, we also extend the normal concrete test scenario by adding failure traces into this concrete test scenario. Table 8.2 shows an example of additional designed abstract test scenarios for failure testing, TS-2.

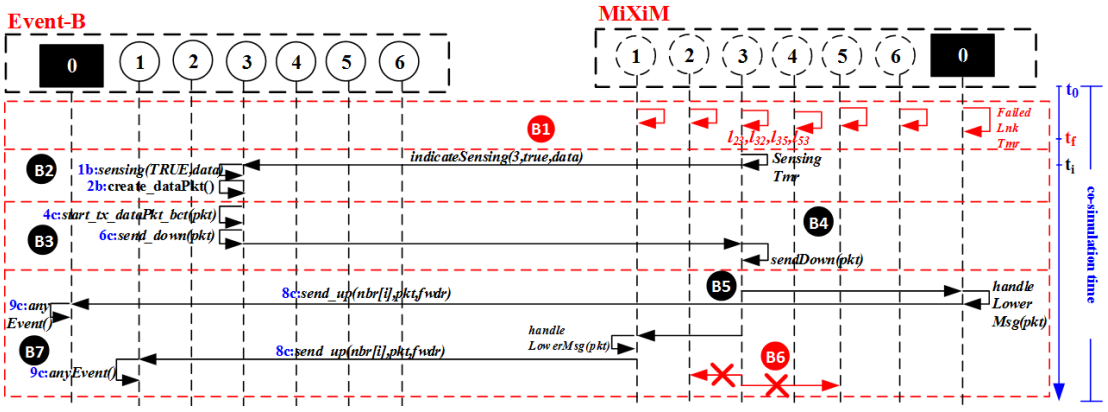
Table 8.2: Abstract failure test scenarios *TS-2*

TS#	TS description	Requirement ID
TS-2 Some Links Surrounding Transmitting Node lose the connection		
TS-21	Validating sensor nodes' and a sink's functionality after some links fail*	**FUN-1, FUN-2**
TS-22	Validating broadcasting mechanism after some links fail*	**FUN-3a**
TS-23	Validating neighbourhood discovery after some links fail **	**FUN-3a**
TS-24	Validating link quality estimation mechanism after some links fail **	**FUN-3b**
TS-25	Validating route broadcast mechanism after some links fail **	**FUN-3c**
TS-26	Validating parent selection mechanism after some links fail **	**FUN-3d**
TS-27	Validating data packet forwarding mechanism after some links fail **	**FUN-3e**

* -*SensorApp* and ** - *MintRoute*

Figure 8.5 shows one of concrete test scenarios in scenarios TS-2 that is test scenario TS-21 describing partial link failure of node 3 as in Figure 8.4(a). This test scenario extends the abstract scenario described in Figure 8.3(b) with link failure injection. Steps (B2), (B3) and (B7) represents the Event-B traces that corresponds to steps (A1), (A2) and (A3) in an abstract test scenario shown in Figure 8.3(b) respectively. Red elements in this diagram indicate the event of failure as shown in steps (B1) and (B6).

As can be seen in Figure 8.5, we indicate some links surrounding the sender to be failed in order to introduce a packet loss during transmission. In this scenario, links l_{23}, l_{32}, l_{35} and l_{53} fail at time t_f before node 3 starts sensing and transmitting a packet at time t_i. This causes the neighbour nodes (2 and 5) of node 3 to not receive such a transmitted packet. The number of lost packets will be recorded and used to calculate the link quality ratio which is very low at the end. In test case *TC-21* in Table 8.4 corresponding to this test scenario, for example, failed time t_f was set to 230s to fail the corresponding links before the second sensing cycle of node 3 (t_i=240s). Note that we configured *SensingTmr* to be 120s to indicate each node to sense data periodically every 120 seconds.



(a)

B1	*failedLnkTmr* indicates the timer for the link failure. This timer activates each virtual node to get failed link information.
B2	*SensingTmr* activates each node controller (e.g. node 3) to sense data. Shared and local events *sensing* and *create_dataPkt* are used to sense and create a data packet respectively.
B3	Shared event *send_down* passes the packet information to the corresponding virtual node in the MiXiM environment. This enables the virtual packet to be created
B4	Then, the virtual packet is transmitted down (*sendDown()*) to the lower layer and finally to the wireless channel.
B5	Next, the transmitted virtual packet is sent up from wireless channel to the neighbour node (*handleLowerMsg()*).
B6	This packet is not *sent_up* to the neighbour nodes who have a failed link.
B7	Shared event *send_up* passes the receiving information from the alive receiving node in MiXiM to the corresponding controller node in Event-B.

(b)

Figure 8.5: Testing scenario TS-2 initiating partial links (of node 3) to be failed (corresponding to Figure 8.4(a))

8.3.5 Designing and Creating Test Cases

The achieved testing scenarios provide us the guideline to design and create executable test cases with its corresponding test data that will be configured in MiXiM's configuration file of our co-simulation framework. As a result of this, functional test cases are derived from testing scenario *TS-1*, the normal transmission and calculation with the perfect network as shown in Table 8.3. For failure and recovery test cases, test scenarios *TS-2* to *-6*) are considered to derive these test cases. Table 8.4 demonstrates an example of failure test cases that is for test scenario *TS-2*, partial link failures. These test cases are the most important test cases especially for the MintRoute protocol. As this protocol establishes and maintains the route tree based on the various link qualities. Thus, most of the test cases in this table are related to monitoring the quality of each link in the network and validating the dead neighbour detection mechanism operated by the MintRoute protocol. Furthermore, as can be seen in Tables 8.3 and 8.4, the specified test cases are also referred to test scenarios that are mapped back to requirements in order to evaluate which requirements have been validated. A test scenario may have one or many test cases associated to it. For example, test scenario TS-24 in Table 8.4 corresponding to test the link quality estimation mechanism for the MintRoute, contains three associated test cases: detecting lost transmitted route packets from failed links, testing sentEst records and checking liveliness calculation.

Table 8.3: Designed functional test cases for test scenarios *TS-1*

TS#	TC#	TC description	Pass/Fail
TS-1: Normal Scenarios			
TS-11	**TC-11**	Validating sensor nodes' and a sink's functionality*	Pass
TS-12	**TC-12**	Validating beacon broadcasting mechanism*	Pass
TS-13	**TC-13**	Validating neighbourhood discovery**	Pass
TS-14	**TC-14**	Validating link quality estimation mechanism**	Pass
TS-15	**TC-15**	Validating route broadcast mechanism**	Pass
TS-16	**TC-16**	Validating parent selection mechanism**	Pass
TS-17	**TC-17**	Validating data packet forwarding mechanism**	Pass

* -*SensorApp* and ** - *MintRoute*

In order to enable our achieved test cases to be executable, we apply the MiXiM's testing methodology in which multiple instances of running experiments (testing scenarios) are created with the different parameters configured in the configuration file to implement specific test cases. In this work, each instance of test cases is created manually based on main criteria including different number of nodes with different network topologies (from 4 to 10 nodes with 1-hop to 4-hop network) and various periods to estimate link quality (e.g. the link quality will calculated every 1, 2 or 3 beacon transmission cycles). An example of creating instances of test cases is shown in Table 8.5. The achieved test cases are expanded in detail that contains test scenario ID (TS), test case description

Table 8.4: Designed failure test cases for test scenarios *TS-2*

TS#	TC#	TC description	Pass/Fail
TS-2: Some Links Surrounding Transmitting Node lose the connection			
TS-21	**TC-21**	Detecting lost transmitted data packet from failed links *	Pass
TS-22	**TC-22**	Detecting lost transmitted beacon packet from failed links**	Pass
TS-23	**TC-23**	Testing Cost Estimation after some link fails**	Pass
TS-24	**TC-24.1**	Detecting lost transmitted route packets from failed links**	Pass
	TC-24.2	Testing *sentEst* record at cycle 3**	Pass
	TC-24.3	Checking liveliness**	Pass
TS-25	**TC-25**	Detecting dead neighbours**	Pass
TS-26	**TC-26**	Checking Choose Parent Module**	**Fail**
TS-27	**TC-27**	Detecting lost forwarded data packet from failed links *	Pass

* -*SensorApp* and ** - *MintRoute*

(TC description), configuration data, pre-condition, test steps and expected results. The expected result is defined by using the ProB model animator and checker. For a given input of test case, we use ProB to animate the Event-B controller and return a result. This result is then recorded as an expected output of the test case. Table 8.5 shows an example of instances of test cases *TC-11* and *TC-23*.

Considering test cases *TC-11* in Table 8.5, this test case is a functional test case that aims to validate the normal sensing and broadcasting mechanism for SensorApp protocol with a perfect network. We design the sample test data to reveal node 3's sensing functionality and transmission mechanism contained in the network topology as shown in Figure 8.4. This is because node 3 has a sink as one of its neighbours so that we can reveal the receiving functionality at the sink node. Whereas test case *TC-23* aims to perform failure testing by injecting some link failure during co-simulation in order to drop the link quality ratio of the failed link below the threshold. We indicate links l_{23}, l_{32}, l_{35} and l_{53} to be failed at time 230s before a sink starts broadcasting a second beacon packet at time 240s. Note that, the periodic broadcast beacon time is set to 120s. This means that a sink broadcasts a beacon every 120 seconds. This causes node 3 not to receive this second broadcast beacon from its neighbour nodes (2 and 5) and leads node 3 to increment the number of lost packets (*missed*) from neighbours 2 and 5 in its neighbour table ($missed = \{((3 \mapsto 2) \mapsto 1), ((3 \mapsto 5) \mapsto 1)\}$). After performing the link quality estimation period, the reception ratio of links between node 3 and these neighbour nodes is calculated to 50% (as it is estimated every two broadcast beacon packets, EST_PERIOD=2).

Table 8.5: Examples of test case instances of *TC-11* and *TC-23*

TS#	TC#	TC Description	Configuration (Test Data)	Pre-condition	Test Steps	Expected Results
TS-11	TC-11	- Validating the functionality of sensor nodes and a sink. - Validating the broadcasting mechanism	sensing period: 120s Number of nodes: 7 sink id: 0	Initialising co-simualtion	1. At time 120s node 3 senses data from environment. 2. Node 3 creates a data packet and broadcasts this packet to its neighbours	All node 3's neighbours including a sink (0, 1, 2 and 5) must receive a transmitted data packet.
TS-23	TC-23	Testing Cost Estimation after some link fails	beacon broadcast period: 120s number of node: 7 sink id: 0 EST_PERIOD: 2	The beacon has been broadcast to all nodes in the network for the first one cycles.	1. At time 230s, link 3-2, 2-3, 3-5 and 5-3 fails (to introduce the second beacon transmitted to node 3 from nodes 2 and 5 to be lost.) 2. From time 230s, the beacon is broadcast to all nodes in the network. 3. Each node calculates the reception ratio. 4. The simulation finishes at time 350..	Node 3 must not receive a broadcast beacon from nodes 2 and 5 as their links to these nodes are broken. The result of neighbour table recorded by node 3: received={((3↦0) ↦2),((3↦1) ↦2),((3↦2) ↦1),((3↦5) ↦1)} missed={((3↦0) ↦0),((3↦1) ↦0),((3↦2) ↦1),((3↦5) ↦1)} lastSeqno={((3↦0) ↦100),((3↦1) ↦100),((3↦2) ↦50),((3↦5) ↦50)}

8.3.6 Testing Execution and Results

Our test cases were run with co-models in the FoCoSim-WSN framework within 1400 seconds of simulation time. The simulation executions were generated from the parameters representing different test cases configured in the MiXiM configuration file. This performed eleven sensing cycles. After the simulation completed, we checked the actual test case results recorded in a log file by comparing it with the expected results. Around 95% of the total number of test case results satisfied the expected result. There were no failed results in the test cases of test scenarios *TS-1* and *TS-4* to *-6* (22 test cases in total). However, two test cases of test scenarios TS-2 and TS-3 enabled us to reveal the computation errors occurring in the MintRoute algorithm as shown in Table 8.6.

Table 8.6: Summary of results containing failed test cases

TS#	# of Test Cases		
	Pass	Fail	Total
TS-1	6	0	6
TS-2	8	**1**	9
TS-3	3	**1**	8
TS-4	8	0	8
TS-5	4	0	4
TS-6	4	0	4

Discoveries from failed test cases: the failed test cases enable us to discover killer traces which can detect an error in our formal controller model. We discovered the faults in the parent selection operation during running of the simulation with test cases TC-26 and TC-32 regarding partial and all link failure scenarios respectively.

Considering test case *TC-26* in Figure 8.7(a), we injected links l_{23}, l_{32}, l_{35} and l_{53} corresponding to Figure 8.4(a) to be failed before each node broadcast a route update packet. We expected these failed links to affect the neighbour nodes surrounding the failed link not to be able to receive a route update packet. This led the transmission ratio (*sentEst*) of such a neighbour node in the neighbour table to be zero. This neighbour node must be excluded when each node calculates and selects a parent. However, there was an error message reported on the Groovy console when co-simulation reached this step. We discovered that such an error came from the wrong guard in event *cal_PCost_le*. Ideally, this event should be enabled when there is a neighbour node who has either the reception (*receiveEst*) or transmission (*sentEst*) ratios less than the quality threshold. In an incorrect model, this condition is implemented as the conjunction of two guards (as guards*@g6* and *@g7* in Figure 8.6(a)) in which both guards have to be true. This implements the wrong semantic which means that this event will operate if both ratios are less than the threshold. This led us to correct this event by combining two guards and using the disjunction predicate to check both ratios (as *@g6* in Figure 8.6(b)).

@g6 receiveEst(x ↦*nb*) < *th* @g7 sentEst(x ↦*nb*) < *th*	@g6 receiveEst(x ↦*nb*) ≥ *th* ∨ sentEst(x ↦*nb*) ≥ *th*
(a)	**(b)**

Figure 8.6: (a) incorrect guards in *cal_PCost_le* and (b) the guard after correction

On the other hand, test case *TC-32* in Figure 8.7(b) enabled us to discover a missed event. This test case injected all links surrounding the transmitting node like node 3 (as illustrated in Figure 8.7(b)) to be failed before transmitting a packet. This led to both link quality ratios of the failed link to drop below the threshold. When the simulation reached to a parent selection step, this node could not identify its parent and led the thread of node 3's controller to stop executing. To correct this, we added new event *choose_parent_skip* to manage the case that there are no suitable neighbours to be a parent

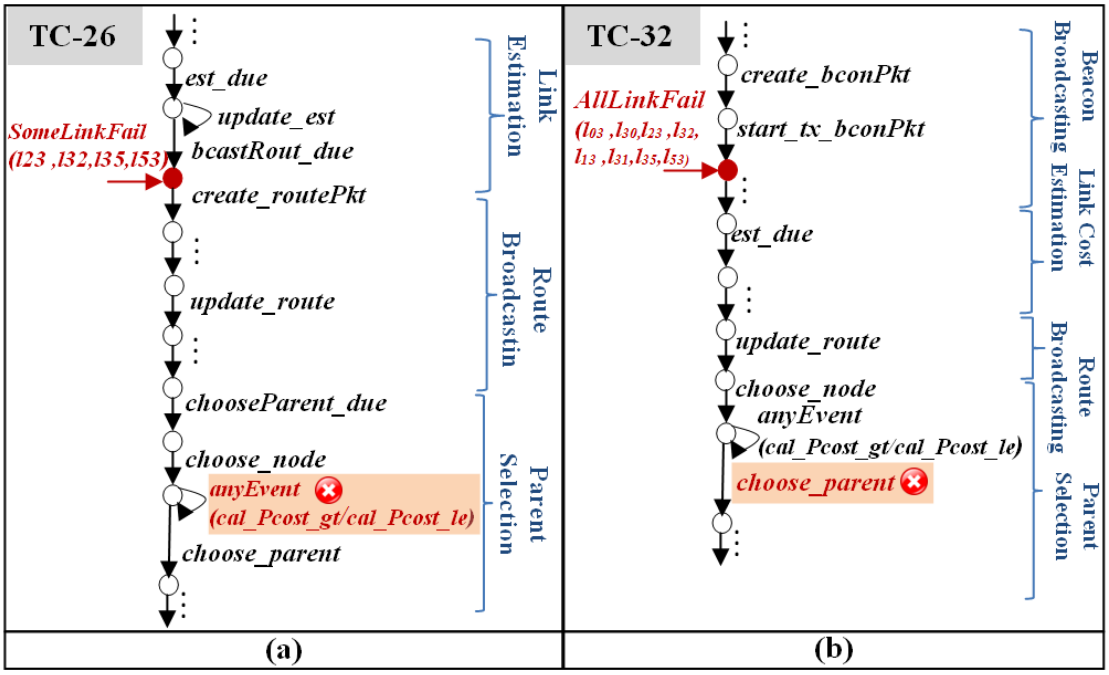


Figure 8.7: "Killer Traces"of test cases (a) *TC-26* and (b) *TC-32*, discovering the error along the trace execution

8.4 Discussion

Based on the experience we have gained from this work, we confirmed the value of combining experimental simulation testing and MBTT approaches. Simulation creates "macroscopic testing views" in which our verified formal model can be executed coarsely with the realistic and stochastic physical environment. For example, we use Signal-to-Noise Ratio (SNR) to model transmission errors in the decider and intentional interference determining packet loss. Random coarse- and large-scale testing scenarios,

generated automatically by simulation environments, enable the non-functional requirements regarding network performance to be analysed and evaluated. Unlike this, the MBTT technique provides "microscopic testing and debugging views" in which the executable formal model can be validated finely through various test cases. Different and specific test scenarios enabling validation coverage created from event traces, the representation of system behaviours and functional scenarios, can validate the correctness of formal model behaviours. Fault injection test cases through simulation can improve test effectiveness which creates "killer traces" to reveal the absent or erroneous constraints and events. Furthermore, validation coverage enabled by MBTT approach can influence simulation testing techniques to be improved.

As the testing framework proposed in this work supports the semi-auto generation from the master that contains traces of Event-B model controller to executable test cases in the configuration file, the mapping between abstract and concrete test scenarios is accomplished manually. Thus, an automatic translation between these two models is required for future work. Open research issue is the potential tool support for MBTT that provides the automatic testing process and test case generation from Event-B model controller in our co-simulation framework.

8.5 Conclusion

This chapter has presented sub-contribution C.4 i.e how MBTT can be used to validate our formal model in the co-simulation environment. The abstract test scenarios are firstly generated automatically in the event trace diagram. This diagram contains the sequence of the events or the event traces of the Event-B controller expressing the scenario or phenomenon of the model behaviours. More concrete test scenarios are created manually from these abstract test scenarios by adding interaction between formal controller model and simulation environment. Test cases are generated at the end and configured in the configuration file in order to drive testing scenarios during co-simulation. This includes the functional, failure and recovery testing which are used to test the valid and invalid traces. As the test scenario/test case is generated from the trace of formal controller that encodes the protocol functionality and algorithms, our results have shown that MBTT benefits the validation coverage process in which the generated test scenarios/test cases can be used to trace back to the requirements. This enables uncovered requirements to be identified and further validated. Furthermore, failing test scenarios enable "killer traces" to detect and debug the absent or erroneous constraints and events. We believe that the validation coverage provided by this MBTT approach can strengthen the V&V including proof-based verification and simulation-based testing techniques provided by *FoCoSim-WSN* framework to enable early detection of defects in WSN systems before the deployment.

Chapter 9

Reuse of the FoCoSim-WSN Co-simulation Framework

9.1 Introduction

The main research contribution is a formal co-simulation approach to increase the quality of WSN development. This contribution takes the form of a formal co-simulation framework (FoCoSim-WSN) which integrates the proof-based formal verification, Event-B, and simulation-based testing techniques, MiXiM as demonstrated in Chapter 7. This chapter describes our sub-contribution C.5, which is usability and reusability of this framework with demonstrating that how they can influence F^S *Co-simulation* development described in Section 7.9.3 by using this framework. Another case study that has a different structure from SensorScope is considered to demonstrate the wider applicability and reusability of our framework. This case study is a wireless sensor and actuator network (WSAN) for the road tunnel monitoring and control system (RTMCS). An overview of this study is provided in Section 9.2 and 9.3 before describing briefly F^S *Co-simulation* development in Section 9.4. Section 9.5 describes the reusability of this co-simulation framework which is classified into three categories: *reusability of formal controller development*, *master implementation* and *simulation implementation*. Furthermore, this section expresses the measurement of Event-B model reuse that is calculated from our developed SensorScope model. This results in the refinement patterns and generic Even-B pattern for controller development which can be used as a guideline for the potential development. These patterns are expressed in Section 9.6. In Section 9.7, we provide the conclusion of this chapter.

9.2 Overview of WSANs

A wireless sensor and actuator network (WSAN) (Xia et al., 2008, 2011; Vicente-Charlesworth and Galms, 2011) is a wireless network system consisting of a number of sensors and actuators to perform distributed sensing and actuation tasks. Generally, there are three essential components in a WSAN: sensors, actuators and sinks as shown in Figure 9.1. Similar to WSNs, a sensor node is an embedded device that normally senses the data from the severe environment. Actuator node, on the other hand, is an electronic device attached to such devices as sprinklers or emergency alarming systems. The actuator triggers an action by using a decision-making process performed when the incoming sensed data from the sensor reveals a risk situation (it is above a safety threshold). The last component of a WSAN is the sink which has a different role in the network depending on the architecture. It may provide decision-making, and activation of the actuator, or monitoring and managing the sensor-actuator network.

The basic architecture for a WSAN can be classified into two types (Xia et al., 2008, 2011; Vicente-Charlesworth and Galms, 2011). Figure 9.1(a) shows the *semi-automated architecture* in which all sensed data is delivered to a sink. This sink includes a decision-making application for diagnosing the level of risk e.g. temperature reading above a safety threshold. Then, the command corresponding to the diagnosis is created at the sink and transmitted to the actuators to perform the appropriate actions e.g. turning on water sprinklers or emergency alarms in order to reduce the risk. The alternative architecture is *automated WSAN* that is demonstrated in Figure 9.1(b). In this architecture, the sensed data is transmitted directly from sensor node to the corresponding actuator node. The actuator node then processes all incoming data and decides whether or not to trigger the appropriate actions. A sink in this architecture is only responsible for monitoring and managing the overall network e.g. sending control and configuration messages in order to maintain network connectivity. The case study that is used in this chapter is based on automated WSAN architecture. The next section describes this case study with its protocol and algorithms.

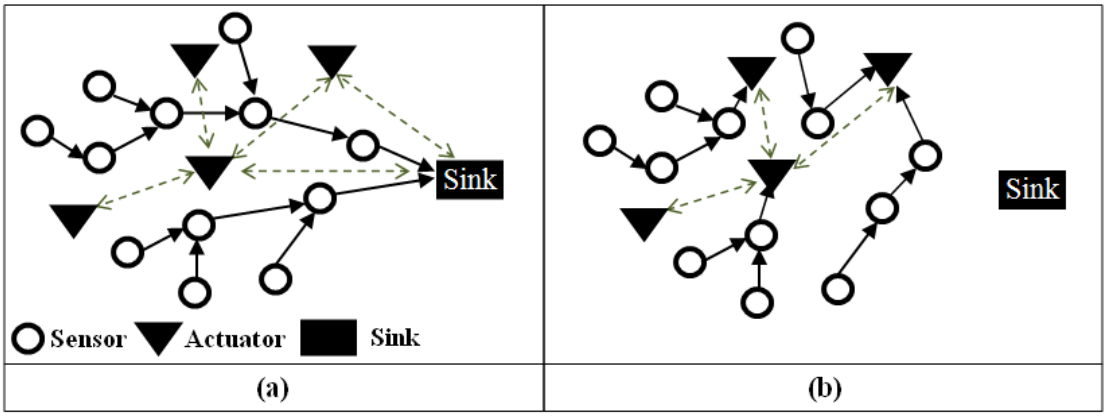


Figure 9.1: Basic WSAN architectures: (a) semi-automated and (b) automated

9.3 Road Tunnel Monitoring and Control System (RTMCS)

In order to demonstrate the usability and reusability of our FoCoSim-WSN framework, we formulate another different case study, a road tunnel monitoring and control system (RTMCS). This case study is created based on Tunnel Control and Monitoring application (Costa et al., 2007). It is an automated WSAN that contains sensor nodes sensing data samples periodically, actuator nodes attached to sprinklers or emergency alarming systems and a sink recording sensed data from all sensor nodes.

Unlike the SensorScope case study, RTMCS is a wireless sensor actuator network (WSAN) based system for monitoring and controlling an emergency situation occurring in a road tunnel. For application and network layer protocols, this case study implements *Sensing-trigger application* and *Ad Hoc On Demand Distance Vector* (*AODV*) routing protocols (Perkins and Royer, 1999; Chakeres and Belding-Royer, 2004) respectively. These protocols are described briefly as follows.

Sensing-trigger application protocol: a group of sensor nodes with their corresponding actuators are placed in the monitoring region as demonstrated in Figure 9.2. Each sensor node in the multi-hop network reports the sensed environment conditions such as temperature to nearby actuators. For example, in Figure 9.2, sensor nodes 2, 4 and 7 reports its sensed data to its nearby actuator node 1. Furthermore, as WSANs are a many-to-many communication network, it is possible that one source node can transmit its sensed data to multiple actuators. For example, sensor node 5 can transmits a sensed packet to actuator nodes 8 and 9. Then, the receiving actuator measures this incoming data. If such data is above a safety threshold, the actuator will determine the occurrence of an emergency situation. This leads the actuator to actuate the safety system (e.g. sprinklers and emergency alert). Furthermore, the sensed data is transmitted to a data logger at a control centre via a data sink as illustrated in Figure 9.2.

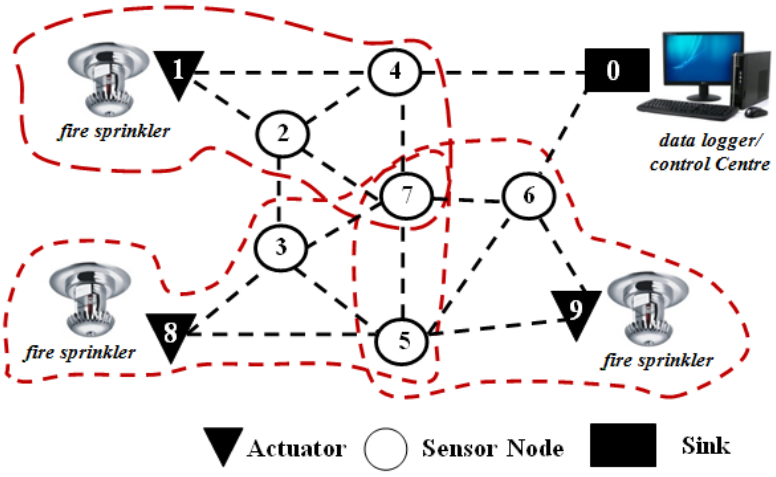


Figure 9.2: Example of RTMCS network

AODV routing protocol: is an on demand algorithm in which the route is constructed when it is required. This routing protocol can be divided into three main steps: *Route Request* (*RREQ*) *broadcast*, *Route Reply* (*RREP*) and *Route Error* (*RERR*) *unicast mechanisms*.

Route Request (RREQ) broadcast mechanism: when a sensing node has data to transmit to an actuator/a sink, it creates and broadcasts a *Route Request* (*RREQ*) to its intermediate neighbours. If the receiving node that is not an actuator/sink, it updates information attached in the received packet to set up a backward path to this source node in the routing table. For example, node 7 in the network topology illustrated in Figure 9.2 has sensed data to transmit to its corresponding actuators 1, 8 and 9. In this case study, we assume that each sensor node has information which nodes are a target actuator node for transmitting a sensed data statically. Thus, sensor node 7 can identify actuator nodes 1, 8 and 9 to be its target destinations for the RREQ transmission. Figure 9.3(a) and (b) demonstrate an example of RREQ transmission between sensor node 7 and actuator node 1. This RREQ is broadcast from node 7 to its neighbours 2, 3, 4, 5 and 6 as shown in Figure 9.3(a). When node 2 that is not the target destination of this RREQ packet receives this packet, node 2 will record the link transmitting such a packet to be a backward path to node 1. Then, node 2 rebroadcasts this packet to its neighbours including actuator 1 as shown in Figure 9.3(b).

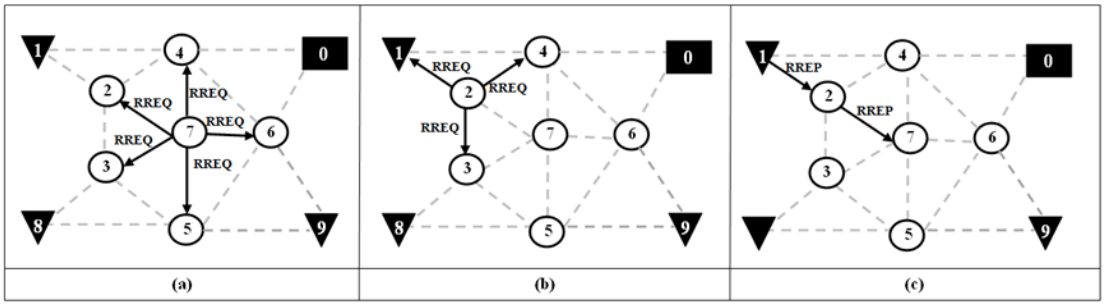


Figure 9.3: Example of RREQ and RREP steps in AODV (a) RREQ for destination node 1 broadcasted by source node 7 to all its neighbours, (b) non-target node 2 rebroadcasts a RREQ packet of node 1 to its neighbours and (c) RREP replied by the destination node 1 to source node 7

Route Reply (RREP) unicast mechanism: if the receiving node is an actuator/a sink, it unicasts a Route Reply (RREP) back to a source node by using the backward path. Along the path transmitting the RREP to the source node, the receiving node records the receiving packet information to set up a forward path to an actuator/a sink. Figure 9.3(c) shows an example of RREP mechanism in which actuator 1 replies RREP to source node 7 via the backward path achieved from RREQ broadcast step.

Note that both control packets mentioned earlier have the same structure. *Sequence number* is included in these packets to determine whether the route information is up-to-date. Furthermore, these packets also contain the hop count information that is

compared with the previous recorded hop count in the receiver's route table. This identifies the better route which is determined by a smaller hop count. This protocol also provides duplicated received packet detection mechanism (for both RREQ and RREP) and discards duplicated packets from the route tree construction process.

Route Error (RERR) unicast mechanism: if there is a broken link during transmitting a data packet along the route, a *Route Error* (*RERR*) is sent to a source node via backward path. The node receiving this RERR invalidates the corresponding link to the unreachable destination in its routing table. When the source node receives the RERR, the unreachable link is also invalidated and the route discovery is reinitiated. For example, in Figure 9.4(a), node 2 forwards a data packet transmitted by source node 7 and detects the failed link to actuator 1 later. This causes RERR packet to be created and transmitted to source node 7 (as in Figure 9.4(b)) in order to mark the route to actuator 1 to be invalid and reinitiate this route if necessary.

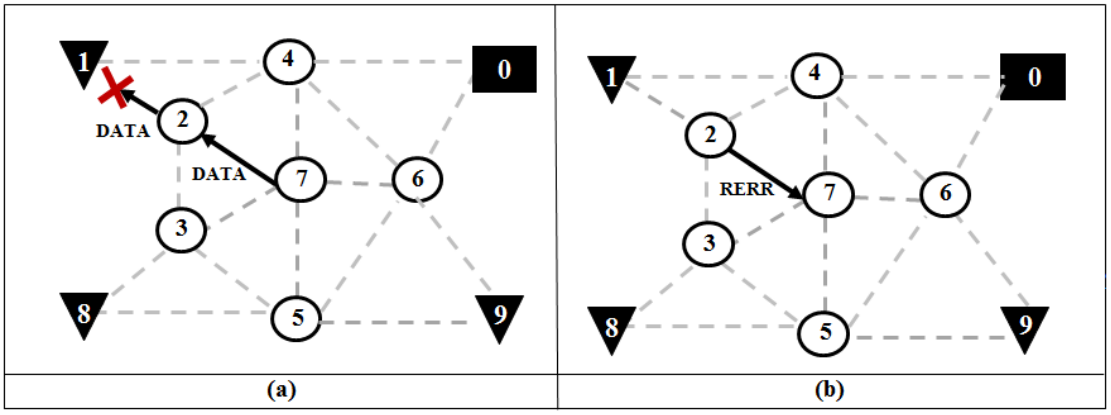


Figure 9.4: Example of RERR step in AODV (a) node 2 discovers failed link after forwarding a data packet of node 1 and (b) RERR transmitted by forwarder node 2 to source node 7 after it discovers the failure

9.4 From-F-through-S-to-C (F^SCo-simulation) Development

As the main objective of this chapter is to demonstrate the reusability of co-simulation development together with the application to F^SCo-simulation development, this section provides an overview of this type of co-simulation development.

F^S Co-simulation development is one technique of co-simulation development proposed in Chpater 7. It is C-style development from F-style through S-style which layers the protocol algorithm for formal verification. Thus, the development technique starts with F-style development that layers the protocol algorithm through refinement steps in Event-B before the verified model is integrated with the environment models in MiXiM. For MiXiM environment model development, the standard based layer (`BaseLayer`) provided by MiXiM can be extended to implement each layer of the protocol stack in order to connect

to the formal controller. However, an alternative way to develop the MiXiM model is to reuse our standard modules `SimpleApp` and `SimpleNet` for communicating between two upper layers as shown in Figure 9.5.

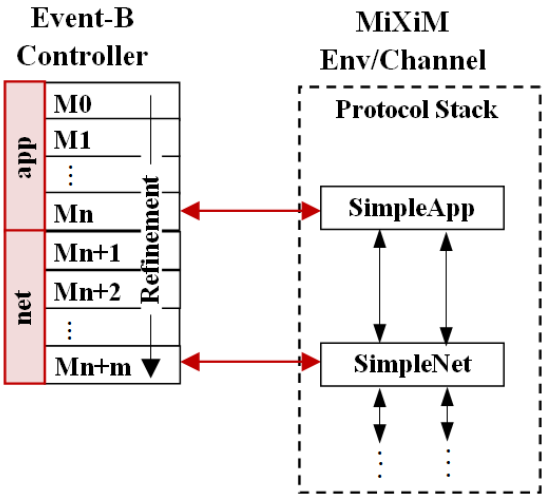


Figure 9.5: Example of F^{S}Co-simulation development of two upper layer protocols

In the following section, we describe this F^S *Co-simulation* development with the reusability of co-simulation development by using the RTMCS case study. By comparing with SensorScope, we discover refinement patterns for controller development that can be reused in F-style development. This also demonstrates with how our extended MiXiM environment and master structure developed for SensorScope can be reused into the RTMCS development.

9.5 Reusability of Co-simulation Development

This section describes how each co-simulation component implementing our previous case study, SensorScope can possibly be reused to support F^S *Co-simulation* development by using RTMCS. We classify the reusability of co-simulation development into three categories: *reusability of Event-B model controller, master implementation and simulation environment.* First of all, we start with considering the reusable formal model for controller development that will be described in Section 9.5.1. This results in the level of reusability from the development of SensorScope to the application of RTMCS modelling. Furthermore, in order to evaluate the effectiveness of this reusability, this section also explains the measurement of SensorScope model reuse. Then, in Section 9.5.2, the reusabiltiy of the master implementation is taken into account. Finally, in Section 9.5.3, the reusability of MiXiM simulation environment model will be considered.

9.5.1 Reusable Formal Models for Controller Development

9.5.1.1 Refinement Patterns for Controller Development

Based on the experience we have gained from the Event-B development described in Section 6.3, we discovered the refinement steps implementing SensorScope suggested the refinement pattern that could facilitate the refinement process of other applications as expressed in Figure 9.6. This refinement pattern provides steps of refinement strategies that are grouped into two parts. The former part ((A) in Figure 9.6(a)) is firstly considered in which the functionality and operation for the specific protocol algorithm is defined, before layering the later part ((B) in Figure 9.6(a)) by defining packet transmission and physical interaction mechanisms via wireless channel which are reused from our previous development (shared events explained in Section 6.7). This influences the benefit that these steps of refinement strategies can be repeated to develop incrementally each layer protocol. Here we demonstrate this benefit with application and network protocol modelling as shown in Figure 9.6(a) and explain it with SensorScope development as an example as illustrated in Figure 9.6(b). We also apply this refinement pattern to RTMCS modelling as shown in Figure 9.6(c). This is described in detail in Section 9.5.1.2.

Application protocol modelling: the application protocol algorithm is layered first by considering functionality (e.g. create and receive packet operations) implemented locally in each node. This can start with modelling the abstraction in which each node creates and transmits a data packet abstractly (e.g *M0* of SensorScope in Figure 9.6(b)) before the neighbour node is considered to implement the forwarding operation (e.g *M1* in Figure 9.6(b)). Then, the sensing environment is taken into account to implement the sensing mechanism (e.g *M2* in Figure 9.6(c)). The packet transmission and physical interaction mechanism for application protocol with simulation environment is also integrated into the final model of the application protocol. Shared events are implemented as a port to interact with physical environment in simulation including *sensing* for sensing mechanism, *send_down* and *send_up* for adjacent layer gate communication (transmitting a packet up and down to a channel).

Network protocol modelling: the step of refinement strategies for application protocol modelling can be repeated on the development in routing protocol. Thus, this begins with layering functionality of routing protocols such as neighbourhood discovery, link quality estimation, route broadcast, and parent selection in MintRoute protocol as expressed in *M3-M5* in Figure 9.6(b). Then, the packet transmission and physical interaction mechanism for network protocol is taken into account. The shared events achieved from the application model can be refined to the final refinement model of the network protocol in order to interact with the physical environment in simulation.

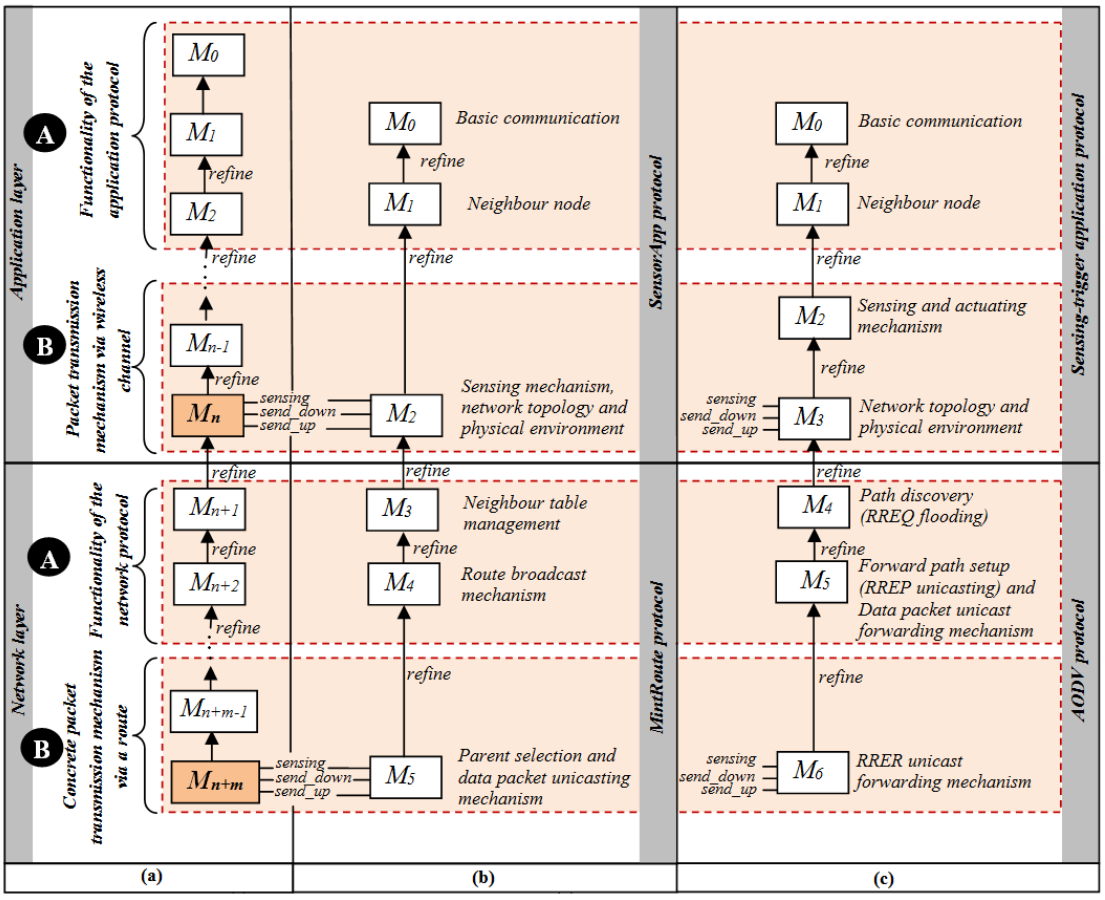


Figure 9.6: (a) Refinement Pattern for WSN development with (b) an example of SensorScope modelling (c) an application of RTMCS modelling

9.5.1.2 RTMCS refinement strategies

Figure 9.6(c) shows the application of this refinement pattern in Figure 9.6(a) to RTMCS case study. The formal model of the RTMCS is given in Appendix C. By comparing this figure with the refinement steps for SensorScope development as demonstrated in Figure 9.6(b), the refinement steps of both case study development are slightly different. We have also discovered that RTMCS requires the same basic communication and channel mechanism as same as SensorScope. Therefore, we start with the implementation of the basic communication before introducing the detail implication of implement application protocol e.g. actuation, the different routing algorithm and so on.

In RTMCS refinement, all elements of abstract models *M0* and *M1* implemented in SensorScope are reused at the same level of RTMCS. We only implement a small change in these refinement models in which we replace a single sink buffer with multiple destination (sink and actuator) buffers. However, we reuse without change the sensing mechanism in model *M2* of SensorScope. We add an actuation event in this model which triggers when the sensed data is raised above a safety threshold. However, we separate the network topology and channel mechanism implemented in model *M2* of

SensorScope to be implemented in the later refinement model in order to manage and simplify the refinement steps and manage the model more efficiently. Later refinement levels *M4* and *M5* layer AODV protocol steps. Model *M4* implements RREQ flooding mechanism before RREP unicasting operation and data packet forwarding mechanism via the route are introduce in model *M5*. Finally, RERR transmission is developed in the final refinement model (*M6*). As can be noticed in this figure, the packet transmission and physical interaction with simulation environment is implemented into the final model of both protocols (*M3* and *M6*) respectively.

9.5.1.3 Reusability of Event-B Model

Based on RTMCS refinement strategies mentioned in the previous section, we reuse the formal models representing SensorApp and MintRoute protocols for SensorScope to implement Sensing-trigger application and AODV routing protocols for RMTCS respectively.

In Event-B development, we consider events as providing functionality of the system. Basically, an event in Event-B defines the dynamics of the transition system that normally modify state variables. Consequently, we consider the level of reusability of our Event-B SensorScope model based on this dependency between reusable event and its modified variables. To accomplish this, we consider which events in SensorScope can be reused to form the functionality of RTMCS system. The selected events enable us to indicate related variables and invariants. Here we briefly describe the dependencies between events and variables before discussing the level of reusability.

Dependencies between events and variables: in Event-B modelling, the dependencies between events and variables is defined and considered in the shared event decomposition and composition technique as proposed and demonstrated in Butler (2008). Event e depends on variable v meaning that event e reads and assigns variable v. For example, Figure 9.7 illustrates the dependencies between event and variables in machine *MN1* before it is decomposed into sub-machines *M1* and *N1*. As can be seen in this Figure 9.7(a), events *E1* and *E3* depend on variables x and y respectively. In terms of Event-B this means that variable x appears in guards or actions of *E1* and so forth for variables y and event *E3*. Event *E2* depends on these two variables. Figure 9.7(b) demonstrates sub-machines *M1* and *N1* that are split from machine *MN1*. These sub-machines can be viewed as an independent sub-component that has its own local variables and events. Variables x and y in machine *MN1* are partitioned into two sub-machines: *M1* and *N1* respectively as shown in Figure 9.7(b) Event *E1* corresponding to variable x becomes a local event in machine *M1* and event *E3* corresponding variable y becomes a local event machine *N1*. These two variables also suggests event *E2* to be a shared event.

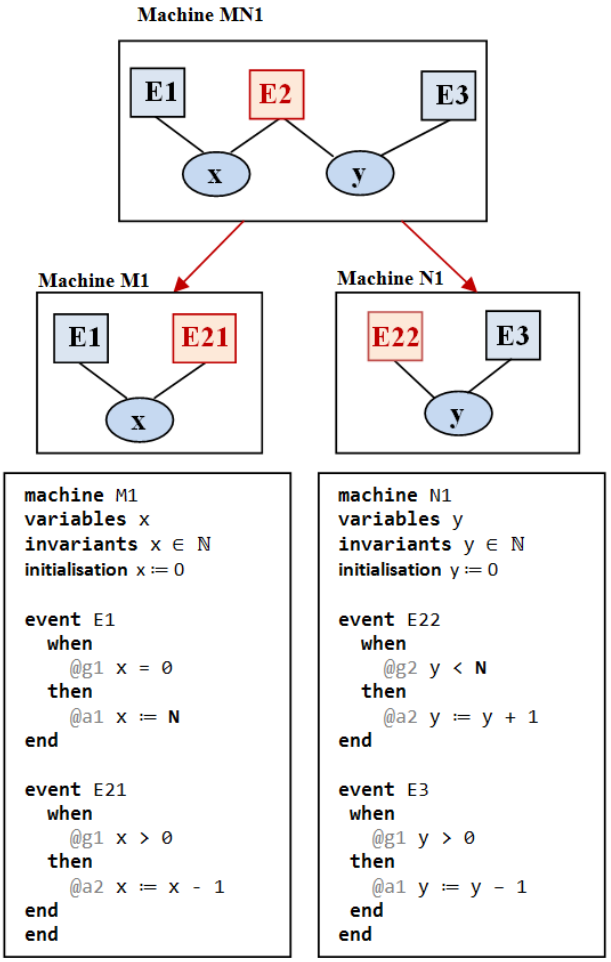
```
machine MN1
variables x y
invariants x ∈ ℕ ∧ y ∈ ℕ
initialisation x ≔ 0 || y ≔ 0

event E1
  when
    @g1 x = 0
  then
    @a1 x ≔ N
end

event E2
  when
    @g1 x > 0
    @g2 y < N
  then
    @a1 x ≔ x-1
    @a2 y ≔ y + 1
end

event E3
 when
   @g1 y > 0
 then
   @a1 y ≔ y - 1
 end
end
```

(a)



(b)

Figure 9.7: An example illustrating the dependency between events and variables (a) machine MN1

To enable the level of reusability, we consider the dependency between events and variables to identify reusable events and theirs corresponding variables and invariants from SensorScope model. This results in the generic Event-B pattern to be identified from generic reusable events. We apply the benefit of shared-event decomposition to form our generic Event-B patterns in which each pattern can be viewed as independent subcomponents containing generic reusable events. This generic Event-B patterns are explained in Section 9.6. The following describes the level of reusability.

The level of reusability: we have discovered that our developed SensorScope model enables the level of reusability as described in the previous section. For example, the generic packet transmission and physical interaction mechanism via the wireless channel demonstrated in part (B) of Figure 9.6(a) is totally reused to implement that mechanism in RTMCS model. Consequently, we reuse the events corresponding to this generic mechanism in SensorScope model to implement the common events in RTMCS model. This also enables the related variables and invariants of reused event to be identified. Table 9.1 and 9.2 show an example of mapping of reusable events between these two

case studies at the application and network level respectively. Note that Table 9.2 does not show all events refined from the application model. In order to demonstrate the partial reusability and the refinement of generic mechanism from the application model, some refined events related to these e.g. events corresponding to creating control packet mechanism and shared events *sensing*, *send_down* and *send_up* are shown in this Table.

We classify the level of reusability of our Event-B model into three categories: *reused without modification* (*total reusability*), *reused with modification* (*partial reusibility*), and *not reused* based on the reusability classification proposed by Washizaki et al. (2012). Figure 9.8, for example, shows the level of reusability from the existing development (e.g. SensorScope) to the new development (e.g. RTMCS). This existing development contains Event-B events *e11* -*e15* in which events *e11* -*e13* are reusable events in a new development. Event *e13* is assumed to be a total reusable event as it provides the generic pattern e.g. the packet transmission, wireless communication and channel mechanism that can be totally reused. Whereas events *e11* and *e12* are a partial reusable element which provides the partial pattern which can be applied to the new project e.g. we adopt event sink *create_bconPkt* in SensorScope project to implement event *create_rreqPkt* in RTMCS project by modifying the guard regarding BEACON packets to be RREQ packets. Finally, events *e14* and *e15* are the elements that cannot be reused as this is because they implement the requirement that differs from the new development e.g. the MintRoute operations comparing to the AODV operations.

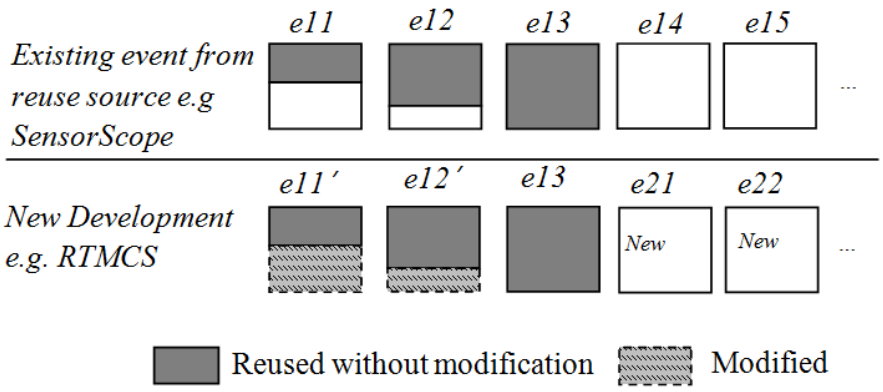


Figure 9.8: The level of reusability

Calculation of Reuse Ratio: in software engineering perspective, software reuse is commonly measured as a ratio of reused code to the total amount of code in a given system (Banker et al., 1993; Washizaki et al., 2012). A high reuse rate indicates that the original developed project supports the high reusability property by meaning that the reusable source is much easier to be reused/adopted to the new development.

Based on the study of Washizaki et al. (2012), we adopt their reuse rate calculation to measure the reusability of our Event-B model as can be shown in (9.1). This reuse rate is measured based on the division between the number of events totally reused without

Table 9.1: Reusability of events in Event-B application layer model between *SensorScope* and *RTMCS* in terms of reusability

SensorScope			RTMCS		
Refinement levels	**Description**	**Events**	**Events**	**Description**	**Refinement levels**
Application Layer Protocol					
M0 (Basic Communication)	creating packets	create_pkt (ctl)	create_pkt (ctl)	creating packets	**M0 (Basic Communication)**
	receiving packets at sink	sink_recv_pkt (ctl)	dest_recv_pkt (ctl)	receiving packets at destination	
M1 (neighbour node)	creating data packets	create_dataPkt ⊑ create_pkt (ctl)	create_dataPkt ⊑ create_pkt (ctl)	creating data packets	**M1 (neighbour node)**
	creating control packets	create_controlPkt ⊑ create_controlPkt (ctl)	create_controlPkt ⊑ create_controlPkt (ctl)	creating control packets	
	transmitting a packet to middleware	start_tx (ctl)	start_tx (ctl)	transmitting a packet to middleware	
	indicating neighbours	indicate_neighbours (env)	indicate_neighbours (env)	indicating neighbours	
	receiving packets at sensor nodes	receive_pkt (ctl)	receive_pkt (ctl)	receiving packets at sensor nodes	
	receiving duplicated packets at sensor nodes	receive_dup_pkt (ctl)	receive_dup_pkt (ctl)	receiving duplicated packets at sensor nodes	
	receiving packets at sink	sink_recv_pkt ⊑ sink_recv_pkt (ctl)	dest_recv_pkt ⊑ dest_recv_pkt (ctl)	receiving packets at sink/actuator	
M2 (Sensing mechanism and Environment)	**indicating sensing from environment**	**start_sensing (env)**	**start_sensing (env)**	**indicating sensing from environment**	**M2 (Sensing and Actuating mechanism)**
	data sensing by each sensor node	**sensing (shd)**	**sensing (shd)**	**data sensing by each sensor node**	
			actuating	activating the safety system	
			no_actuating	not activating as received data below the safety theshold	
(Network Topology and Channel mechanism)	**sending a packet down to a channel**	**send_down ⊑ send_down (shd)**	**send_down ⊑ send_down (shd)**	**sending a packet down to a channel**	**M3 (Network Topology, Physical Environment and Channel mechanism)**
	indicating specific neighbour	assign_fowarder (env)	assign_fowarder (env)	indicating specific neighbour	
	indicating all neigbours	find_neighbours (env)	find_neighbours (env)	indicating all neigbours	
	sending a packet up to neighbour nodes	**send_up ⊑ indicate_neighbours (shd)**	**send_up ⊑ indicate_neighbours (shd)**	**sending a packet up to neighbour nodes**	

total reusability **partial reusability** **not reused**

Table 9.2: Reusability of events in Event-B network layer model between *SensorScope* and *RTMCS* in terms of reusability

SensorScope			RTMCS		
Refinement levels	**Description**	**Events**	**Events**	**Description**	**Refinement levels**
Network Layer Protocol					
M3	creating a beacon packet at a sink	create_bconPkt ⊑ create_controlPkt (ctl)	create_rreqPkt ⊑ create_controlPkt (ctl)	creating a RREQ packet	**M5**
(Neighbour Table Management)	**sending a packet down to a channel**	**send_down ⊑ send_down (shd)**	**send_down ⊑ send_down (shd)**	**sending a packet down to a channel**	**(RREQ Broadcasting)**
	sending a packet up to neighbour nodes	**send_up ⊑ send_up (shd)**	**send_up ⊑ send_up (shd)**	**sending a packet up to neighbour nodes**	
	adding new neighbour in neighbour table	add_newEntry (ctl)	add_bwdRouteEntry (ctl)	adding backward paths from source to destination in neighbour table	
	updating existing neighbour in neighbour table	update_nbr (ctl)			
M4	creating a route packet at each nodes	create_routePkt ⊑ create_controlPkt (ctl)	create_rrepPkt ⊑ create_controlPkt (ctl)	creating a RREP packet at a sink	**M6**
(Route Broadcast Mechanism)	**sending a packet down to a channel**	**send_down ⊑ send_down (shd)**	**send_down ⊑ send_down (shd)**	**sending a packet down to a channel**	**(RREP and data pakcet unicasting mechanism)**
	sending a packet up to neighbour nodes	**send_up ⊑ send_up (shd)**	**send_up ⊑ send_up (shd)**	**sending a packet up to neighbour nodes**	
	updating route information into neighbour table	update_route (ctl)	add_fwdRouteEntry (ctl)	updating forward paths from source to destination into neighbour table	
M5	**data sensing by each sensor node**	**sensing (shd)**	**sensing (shd)**	**data sensing by each sensor node**	**M7**
(Parent Selection and data packet unicasting Mechanism)	**sending a packet down to a channel**	**send_down ⊑ send_down (shd)**	**send_down ⊑ send_down (shd)**	**sending a packet down to a channel**	**(Route Error (RRER) broadcasting mechanism)**
	sending a packet up to neighbour nodes	**send_up ⊑ send_up (shd)**	**send_up ⊑ send_up (shd)**	**sending a packet up to neighbour nodes**	
	choosing node to indicate a parent	choose_node (ctl)	create_reerPkt ⊑ create_controlPkt (ctl)	creating a RRERpacket at a sink	
	calculating path cost	cal_PCost_gt (ctl)	invalidate_links (ctl)	invalidating failed links	
		cal_PCost_gt_skip (ctl)			
		cal_PCost_le (ctl)			
	choosing a parent	choose_parent (ctl)			
		choose_parent_skip (ctl)			

total reusability **partial reusability** **not reused**

modification and the number of events adopted from the original existing development to clone/implement the corresponding elements in the new development. Adopt means any terms of use including with and without modification. As a result of this, the total reuse rate of an example illustrated in Figure 9.8 is 1/3 (=33%). This is because there is one event which is totally reused of three events adopted from the original existing development to implement the element in the new development.

$$Total\ Reuse\ Rate = \frac{totalReuseEvents}{adoptedEvents} \tag{9.1}$$

where

- *totalReuseEvents* is a total number of total reused events in the new development
- *adoptedEvents* is a total number of adopted events in the new development

Therefore, the partial reuse rate is calculated as illustrated in (9.2). It is calculated as $100\ -\ total\ reuse\ rate$. The partial reuse rate of an example shown in Figure 9.8. is 2/3 (66.67%).

$$Partial\ Reuse\ Rate = \frac{adoptedEvents\ -\ totalReuseEvents}{adoptedEvents} \tag{9.2}$$

Furthermore, we are also interested in the rate of new events introduced to the new development. This rate is conducted in (9.3). As a result of this, the new rate of an example illustrated in Figure 9.8 is 2/5 (40%).

$$New\ Rate = \frac{newEvents}{adoptedEvents\ +\ newEvents} \tag{9.3}$$

where

- *newEvents* is a total number of new events introduced to the new development

Measurement of Event-B Model Reuse: based on the resuability of events between two development as shown in Table 9.1 and 9.2, we can calculate the reuse rate of each machine as demonstrated in Table 9.3.

We discovered that most elements defined in formal models, machines *M0-M2*, for the application protocol in SensorScope have the high reuse rate. This means that they can be reused easily in the implementation of RTMCS models. Only small changes need to be done in the abstract refinement model (*M0*) for this application layer protocol in which a single sink buffer are replaced with multiple destination (sink and actuator) buffers.

Table 9.3: Reuse rate of events in SensorScope case study

Source Machine (SensorScope)	Total Reuse (1)	Partial Reuse (2)	Adopted (1)+(2)	Target Machine (RTMCS)	New (3)	Total Events (1)+(2)+(3)
M0	3 (75%)	1 (25%)	4	M0	0	4
M1	12 (92.31%)	1 (7.69%)	13	M1	0	13
M2	22 (95.65%)	1 (4.35%)	23	M2-M3	2 (8%)	25
M3	26 (92.85%)	2 (7.14%)	28	M4	2 (6.67%)	30
M4	23 (82.14%)	5 (17.86%)	28	M5	2 (6.67%)	33
M5	23 (82.14%)	5 (17.86%)	28	M6	3 (9.09%)	32

Thus, this affects 75% of this figure to be totally reused. However, in later refinement models *M1-M2*, we achieve an higher total reuse rate. This is because these models contain the generic pattern e.g. sensing, packet transmission and channel mechanisms as demonstrated in Figure 9.6. Consequently, this enables the standard data structure such as sensing buffer and packet structure together with shared events to be totally reused. Furthermore, in model *M2* of RTMCS, there is a small rate of introduced new events (only 8% of new rate as shown in Table 9.3) in which we only add two new events regarding the actuating events to implement triggered events for actuators as already illustrated in Table 9.1.

Considering the total reuse rate for the network protocol level (machines *M3-M5*), we see that each protocol refinement level still has the high total reuse rate. We discovered that most reusable elements affecting the reuse rate are the generic elements refined from the application level model. For example, in machine *M3*, the total reuse rate of this machine is 92.85%. This is because model *M3* refines the generic elements from the application level model *M2*. Detailed evidence for this is given in Table 9.2 which compares the operations of these two network protocols. Furthermore, in Table 9.3, the total reuse rate of AODV refinement model is reduced from 92.85% (in *M3*) to 82.14% (in *M5*). This is because we implement more specific operation of AODV in the concrete model such as *M5*. Consequently, the new rate of these corresponding refinement models are increased from 6.67% to 9.09% as can be seen in Table 9.3. As the functionality of AODV protocol is less complex comparing to that of MintRoute protocol, these new rates are quite low. The details of these introduced new events are shown in Table 9.2.

9.5.2 Reusability of Master Implementation

Figure 9.9 demonstrates the reusable master structure that we discovered from C-style development as described in Section 7.9.3. In this structure, the generic class including main class (`MainClass`), master class (`Master`) and multithread creation (`Task`), socket program (`WSNSocket`) and log file management (`LogMgt`) can be totally reused in the new development. We implement base class `EventBCtl` containing standard methods `getSensedFlg()`, `getSensedData()` and `transmit2Channel()` that

provide general interaction and communication mechanism from Event-B controller model by executing shared events *sensing*, *send_down* and *send_up* to simulation environment via class `Master`.

Based on the reusable master structure demonstrated in Figure 9.9, the master for the specific case study like SensorScope can be implemented by totally reusing the generic classes described above as shown in black rectangle frame in this figure. To implement the trace of Event-B controller for the specific protocol, base class `EventBCtl` can be extended to specific class `SensorscopeCtl` as shown in the blue rectangle frame in Figure 9.9. The specific methods are developed in this class in order to drive the Event-B controller to create event traces representing the specific protocol operation e.g. methods *estLinkRatio()* and *chooseParent()* for link quality estimation and choosing parent mechanisms in MintRoute protocol.

On the other hand, the master for RTMCS is specialised in the same way as SensorScope. This master can be implemented by retaining the generic classes described above. We only extend class `RTMCSCtl` from base class `EventCtl` as shown in red rectangle frame in Figure 9.9. This affects the standard methods in this interface class to be implemented in the extending class RTMCS. We add more specific methods in this class to fulfill the operations of RTMCS. Methods `sensing()` and `actuating()` are implemented to represent sensing and actuating mechanisms for application protocol, whereas methods `sendRREQ()`, `sendRREP()` and `sendRERR()` are developed to represent the operations for AODV routing protocol.

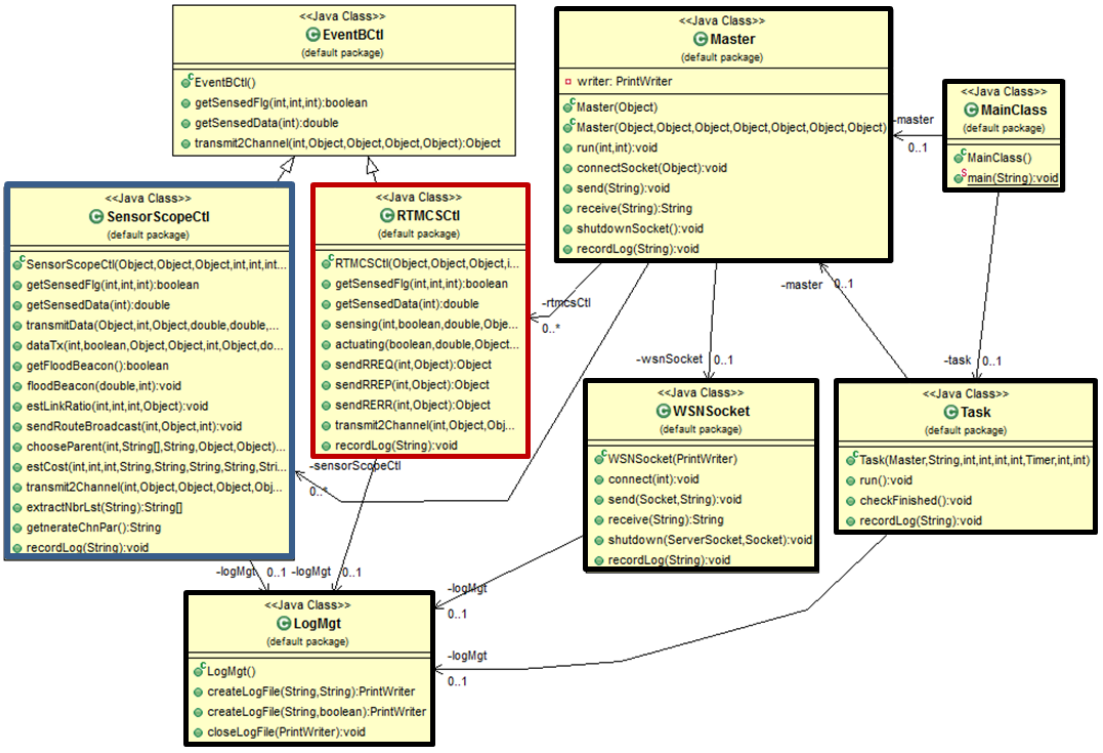


Figure 9.9: Reusable master structure

9.5.3 Reusability of Simulation Environment

As mentioned earlier in Section 7.3, channel parameter `chnPar` synchronising between co-models for two upper protocol layers contains the basic packet structure. Thus, simulation environment in MiXiM is prepared by reusing standard modules `SimpleApp` and `SimpleNet` for communicating between two upper layers. The same structure used in the previous case study as illustrated in Figure 7.9(b) and (c), can be reused to implement co-models for this case study.

9.6 Event-B Pattern of WSN Communication

This section aims to propose the generic pattern and demonstrates its application. As discovered in the previous section, the level of reusability of our SensorScope development in Event-B enables us to abstract the generic pattern. This pattern can be applied to create the common structure instantiation and extended to define some specific for our new case study, RTMCS. To demonstrate this pattern and its application, Section 9.6.1 provides an overview of Event-B pattern concepts. This section introduces the different pattern concepts that can be composed and integrated through the refinement development chain. In Section 9.6.2, we explain briefly the common mechanism for the packet transmission and physical interaction mechanism. We have discovered that this mechanism is a generic mechanism which can be totally reused at the most concrete refinement level of each protocol stack as demonstrated in part (B) of Figure 9.6(a). In Section 9.6.3, we propose the generic pattern structure regarding this total reusability. Section 9.6.4 demonstrates the application of our proposed generic patterns that influences the level of reusability. This research mainly focuses on an independent pattern which can be combined with other patterns. A group of abstract and realising patterns still remains work in progress. However, Section 9.6.4.5 will demonstrate this with an example that can be used as a guideline for the future work.

9.6.1 Event-B Pattern Concepts

Patterns are a general reusable solution that can be commonly used as a guideline to a specification and design problem (Hoang et al., 2011). We define a pattern as the generic specification of Event-B model which can be applied and composed into Event-B model. Therefore, each pattern consists of the generic elements of Event-B model including constants, variables, invariants and a set of their independent events that can be instantiated into the specific Event-B model. Figure 9.10 illustrates how pattern can be applied to the refinement chain of Event-B development. As can be seen in this Figure, the pattern is independent from other patterns and can be composed into Event-B machine at any refinement step. Developers can decide which pattern

they need and where in the refinement chain is applicable for the selected pattern. By applying the pattern, the elements (variables, invariants and events) defined in the applied pattern can be added or composed into the target machine. Either a *single pattern* is selected or *several patterns* are composed to create a target machine as shown in Figure 9.10(a) and (b) respectively. Furthermore, the pattern can be considered as a group of abstract and realising patterns which can map into the abstract and concrete machine in the refinement as demonstrated in Figure 9.10(c). Note that pattern realization is an abstraction relationship that enables stepwise refinement between an abstract pattern and its implementations (concrete patterns).

In the refinement chain, the composed model (prefixed model name with p) can be refined by adding the new events and variable or extending the composed event by adding user-defined guards and actions in order to create the specific operation. To distinguish this model from the composed model, we prefix the name of the refining model containing these user-defined elements with u.

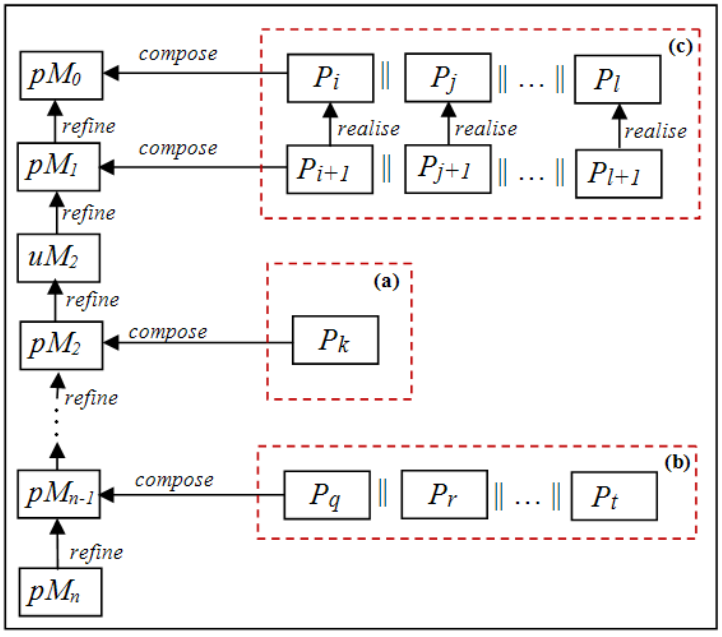


Figure 9.10: Using patterns through the refinement chain of Event-B development

9.6.2 Generic Mechanism for Packet Transmission and Physical Interaction

Our proposed patterns are created based on the generic mechanism for physical interaction and communication as discovered in S- and F-style that contributes to C-style development. Consequently, before demonstrating and explaining the structure of these proposed patterns, this section describes this generic mechanism

In WSN communication, each layer in protocol stack can access a channel by communicating with the adjacent layer via communication interfaces named gates. In S-style development, this mechanism is provided by standard library functions implemented in the MiXiM simulator. Basically, the developer implements the specific protocol including algorithms and calculation for each layer e.g. application and network layer and integrates it with these standard functions in order to enable gate communication between the adjacent layers as shown in Figure 9.11(a).

As in C-style development, we separated each protocol functionality and algorithm to implement as a controller in Event-B as expressed in Figure 9.11(b) and (c) for application and network protocols respectively.

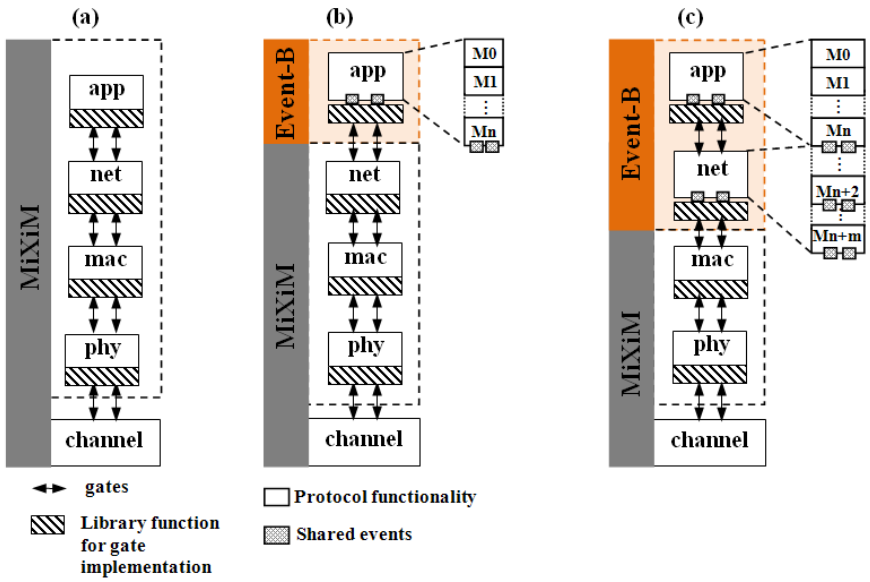


Figure 9.11: The mechanism for interaction and communication between adjacent protocols in protocol stack implemented in (a) S-style and C-style co-modelling with (b) application and (c) routing controllers

Shared events including *sensing*, *send_up* and *send_down* are implemented in the Event-B to connect these formal upper protocol models with the lower protocol model in MiXiM as we demonstrated in F- and C-style development, Chapter 6 and 7 respectively. We implement these shared events for generic mechanism in Event-B controller at the most concrete refinement level of each protocol stack as demonstrated in part (B) of Figure 9.6(a). An example of these concrete refinement level models are machine *M2* and *M5* for application and network layers of SensorScope as illustrated in Figure 9.6(b).

Based on the development explained in Figure 9.11, we have discovered the generic mechanism of Event-B controller modelling for our FoCoSim-WSN as shown in Figure 9.12. This figure contains steps corresponding to the communication steps used in F-style simulation (Figure 6.30) and C-style co-simulation (Figure 7.4). These steps can be described as follows:

(S1) The environment activates each node controller to start sensing data. Shared event *sensing* is used to pass this sensed data to each node controller.

(S2) Each node controller creates a packet (event *creating_pkt*). In the case of a data packet, the sensed data is attached in the create packet. Then, the packet is recorded into a waiting buffer.

(S3) Each node controller selects a packet recorded in a waiting buffer to initiate a packet transmission (event *start_tx*).

(S4) The packet is transmitted from a sender/forwarder down to the lower layer and a channel in MiXiM simulation by using shared event *send_down*.

(S5) Shared event *send_up* is responsible for relaying a transmitted packet from the channel to each receiving node.

(S6) If the receiving node controller is not a target/destination node, the arriving packet will be recorded in the waiting buffer in order to route this packet to the destination. As a result of this, steps (S4)-(S6) will be repeated until the transmitted packet reaches the destination.

(S7) If the receiving node controller is a target/destination node, this received packet will be recorded in the destination buffer.

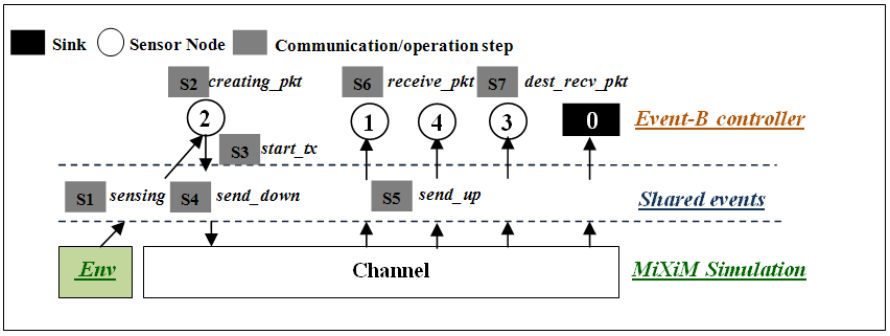


Figure 9.12: the generic physical interaction and communication mechanism of Event-B controller modelling extracted from SensorScope development

9.6.3 Pattern Structure

In order to gain a better understanding of these patterns in detail, a component diagram has been used to express top-level conceptual design. Each component is defined by its interfaces which describe its visible behaviours. Component interfaces are the only access to the component information and functionalities. Components have two related interfaces (UML, 2011a; Sommerville, 2011) as shown in Figure 9.13.

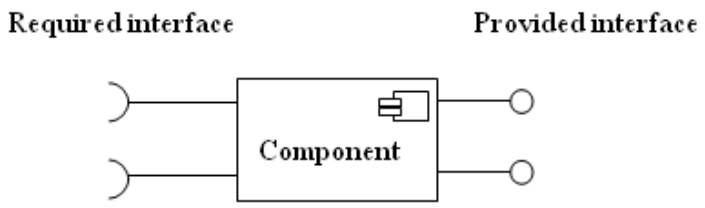


Figure 9.13: Component Interface

- The provided interface defines services provided by the component and is denoted by "a circle notation".
- The required interface specifies what services must be provided by other components in the system and is denoted by "a semi-circle notation".

A component can be viewed as a sub-system which can be combined with other components to create a complex system. In each component, data and services are encoded in component interfaces. The visible behaviour of a component can be realised by services that can be used by other components. The data and services are implemented as variables and methods as demonstrated in Figure 9.14(a). This interface corresponds to implement sensing mechanism that contains two variables: *ctlSenseFlg* and *senseData* for each node's sensed flag and sensed data buffer. Two services are implemented in two methods: *sensing* and *remove_senseBuff* for recording and removing sensed data in/from each node's sense buffer respectively. In Event-B terms, a machine encodes data and services in variables and events respectively as shown in Figure 9.14(b). Consequently, a component interface is equivalent to a machine in Event-B. Furthermore, in Event-B specification context, model composition enables the modelling of interaction of sub-systems in order to generate larger and more concrete specifications. Consequently, in this research, we view our Event-B patterns as component interfaces that are generated into Event-B machines before they are composed to implement interaction of sub-systems by using shared-event composition. For example, events *sensing* and *remove_senseBuff* shown in Figure 9.14(b), are shared events interacting with other sub-systems. Event *sensing* is enabled to receive sensed data from the environment in MiXiM and record it into sense buffer. On the other hand, events *remove_senseBuff* is a shared event that is composed with events *creating_pkt* defined in interface *IPacket*. This refines the operation in event *creating_pkt* that after the sensed data recorded in the sense buffer is attached in a created packet, such data is removed from the buffer. The interaction between these provided and required interfaces is shown in Figure 9.15.

Figure 9.15 demonstrates component *Packet* provides interface *IPacket* and requires interface *ISensingUnit* from component *SensingUnit*. As mentioned above, each pattern is viewed as an interface providing a set of services (functions/events) to accomplish a specific task. As a consequence, two patterns are identified, *PSensingUnit* and *PPacket*

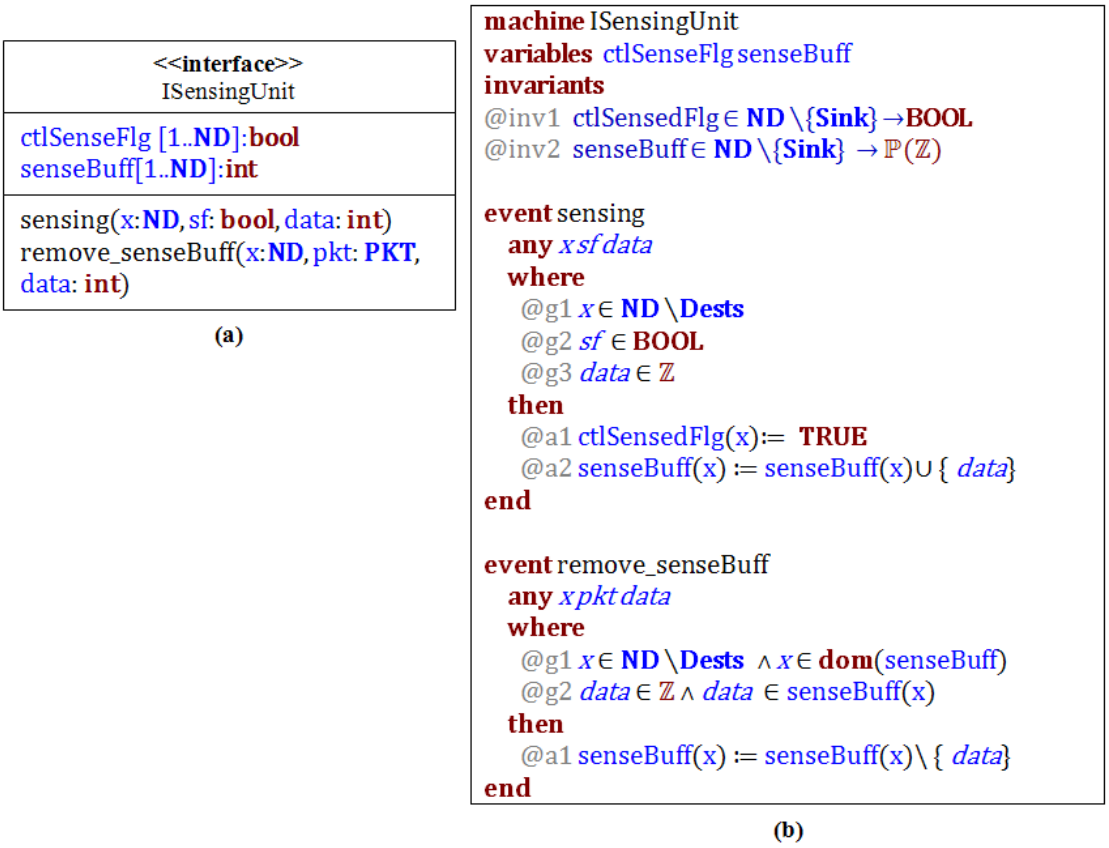


Figure 9.14: An example of: (a) component interface *ISensingUnit* and (b) its equivalent Event-B machine

based on the interfaces defined in this figure. Pattern *PPacket* provides the generic for the packet management. For example, event *creating_pkt* corresponds to packet creating operation that creates a packet and records it into variables *createdPkts*. Furthermore, this event also assigns the source to be an initial forwarder in packet field *pktFwdr* and attaches data to packet field *pktData* respectively. Pattern *PPacket* requires the pattern *PsensingUnit* since this pattern provides the sensed data from sensing environment. The corresponding pattern structure is demonstrated in 9.16. We define $*par*$ to be a generic model parameter which can be instantiated to specific target Event-B model. This will be demonstrated in Section 9.6.4.

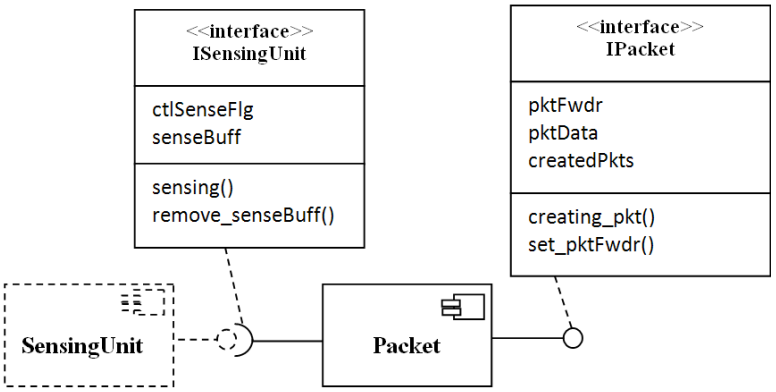


Figure 9.15: An example of provided interface *IPacket* and required interface *ISensingUnit*

```
Pattern PSensingUnit
  includes Context_PSensingUnit
  variables $ctlSenseFlg$  $SenseBuff$
  invariant  $ctlSensedFlg$ ∈ ND ∖ Dests → BOOL
             ∧ $senseBuff$ ∈ ND ∖ Dests → ℙ(ℤ)
  initialization   $ctlSensedFlg$ ≔ (ND ∖Dests) × {FALSE} ||
                   $senseBuff$ ≔ (ND ∖Dests) × {∅}

  sensing ≙  any x sf data
             where x ∈ ND ∖Dests ∧ sf ∈BOOL ∧ data ∈ ℤ
                   ∧ x ∈ dom($ctlSensedFlg$)
                   ∧ $ctlSensedFlg$(x) = sf
             then  $ctlSensedFlg$(x) ≔ sf  ||
                   $senseBuff$(x) ≔  $senseBuff$(x) ∪ {data}
             end
  $remove_senseBuff$ ≙  any x sf data
                        where x ∈ ND ∖Dests ∧ data ∈ ℤ
                              ∧ $senseBuff$(x) ≠ ∅
                              ∧ data ∈ $senseBuff$(x)
                        then
                   $senseBuff$(x) ≔  $senseBuff$(x) \ {data}
                end
End
```

(a)

```
Pattern PPacket
  include Context_PPacket
  variable $pktFwdr$  $pktData$  $createdPkts$
  invariant  $createdPkts$ ⊆ PKT
             ∧ $pktFwdr$ ∈ PKT ⇸  ND
             ∧ $pktData$ ∈  PKT  ⇸ ℤ
  initialization $createdPkts$ ≔ ∅||
                 $pktFwdr$ ≔ ∅ ||
                 $pktData$ ≔ ∅

  $creating_pkt$ ≙  any x des pkt data
                    where x ∈ ND ∖Dests ∧ pkt ∈ PKT
                          ∧ nbh ∈ℤ ∧ sno ∈ ℤ ∧ data ∈ ℤ
                        ∧ x = initialSrcAddr(pkt)
                          ∧  des = ran({pkt} ◁ finalDestAddr)
                        ∧ pkt ∉ dom($pktFwdr$)
                        ∧ pkt  ∉ dom($pktData$)
                    then
                      $createdPkts$≔ $createdPkts$ ∪ {pkt} ||
                      $pktFwdr$ ≔ $pktFwdr$ ∪ {pkt ↦ x} ||
                      $pktData$ ≔ $pktData$ ∪ {pkt ↦ data} ||
                    end

  $set_pktFwdr$ ≙  any  x pkt nbh
                   where   x ∈ ND ∧ pkt ∈ PKT
                         ∧ pkt ∈ dom($pktFwdr$)
                     then
                         $pktFwdr$ ≔ $pktFwdr$  ⊕ {pkt ↦ x}
                    end
end
```

(b)

Figure 9.16: An example of pattern structure (a) *PSensingUnit* and (b) *PPacket*

We have designed Event-B modelling patterns for our FoCoSim-WSN based on the discovered generic mechanism for physical interaction and communication in Figure 9.12. This comprises the following elements: *PSensingUnit*, *PPacket*, *PSend*, *PReceive*, *PND-Buffer* and *PDestBuffer*. The conceptual design of these patterns with their relationships is illustrated by using the component diagram as shown in Figure 9.17. Note that since we view a pattern as an interface, we replace the interface classifier (<<*interface*>>) with the pattern classifier (<<*pattern*>>). The details of these patterns associated with this component diagram have been explained in Appendix D. Here, we show the description of each pattern.

- *PSensingUnit* provides the common patterns for sensing operations of each sensor node including sensing environment data and sense buffer management mechanisms; see further details in Appendix D.1.1.
- *PPacket* provides the common patterns for packet management. This includes creating packet and setting some packet fields such as forwarder (*pktFwdr*) and sensed data (*pktData*); see further details in Appendix D.1.2.
- *PSend* provides the common patterns for transmitting a packet down/up to/from a channel; see further details in Appendix D.1.3.
- *PReceive* provides the common patterns for receiving a packet for the channel at the receivers. This receiving interface provides two common operations: *receiving at the non-destination* and *receiving at the destination*; see further details in Appendix D.1.4.
- *PNDBuffer* provides the common patterns for managing receive and forward buffer at each sensor node. This includes recording/removing data to/from this buffer; see further details in Appendix D.1.5.
- *PDestBuffer* provides the common patterns for managing buffer at destination nodes (e.g. Sink). This includes recording/removing data to/from this buffer; see further details in Appendix D.1.6.

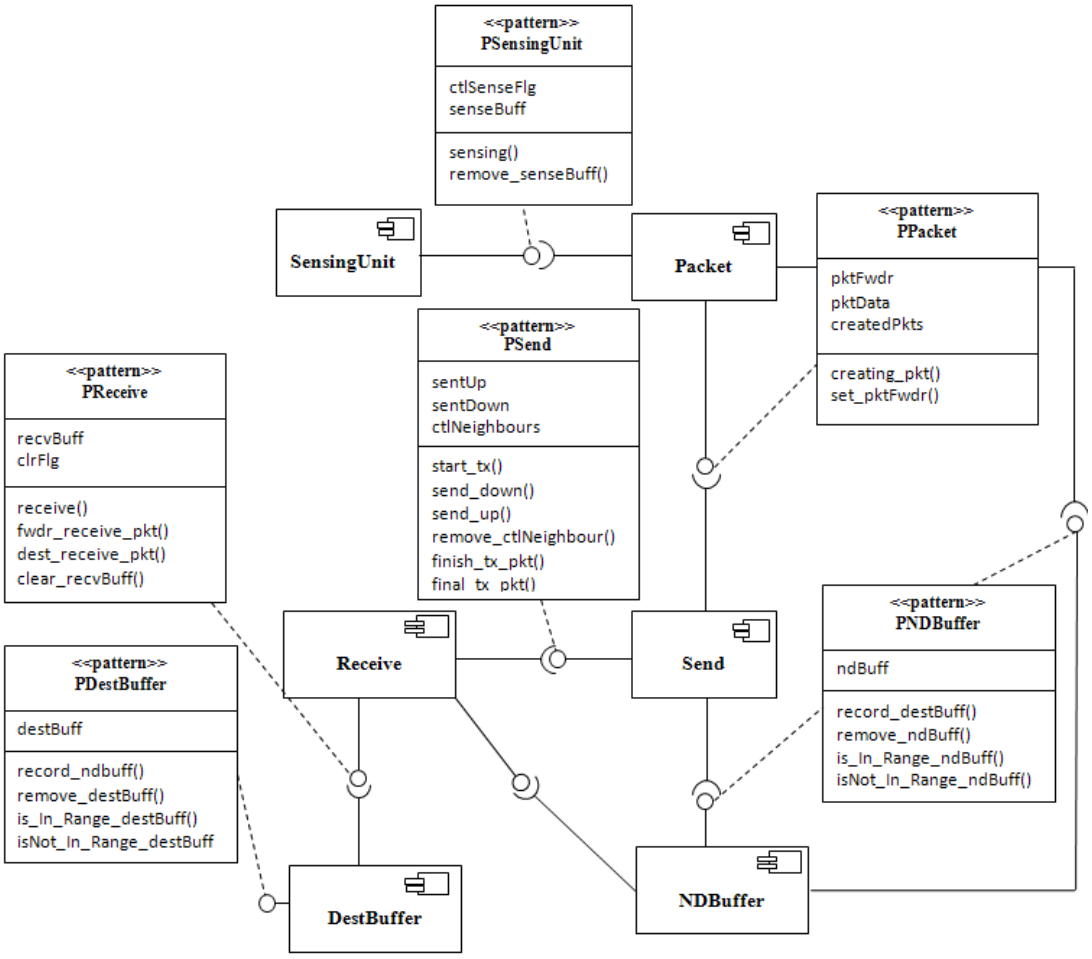


Figure 9.17: Overview of Event-B Modelling Patterns

9.6.4 Applying Patterns

As mentioned earlier, our proposed patterns are independently produced in order to enable the level of reusability. Each pattern can communicate with other patterns by using synchronizing events. In order to create the specific Event-B specification, we apply shared event composition technique provided by Event-B to assemble these generic patterns. Based on our experiences in the development of SensorScope and RTMCS, we have discovered that our proposed Event-B patterns can be applied to create the common structure instantiation required by these two case studies. That is common packet transmission and physical interaction as already mentioned in the previous section. This section shows how each pattern can be instantiated and assembled to enable the generic flow of common communication which is shown in Figure 9.18 that corresponds to Figure 9.12. Furthermore, the specific instantiation for RTMCS by using these generic patterns is also demonstrated in this section.

Tables 9.4 and 9.5 provide an overview of how each pattern can be composed to create the generic flow of common communication as illustrated in Figure 9.18. This results in

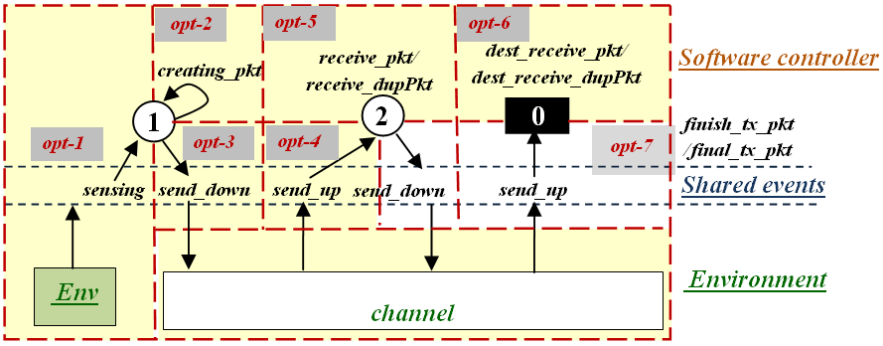


Figure 9.18: The generic flow of common communication mechanism

a common reusable instantiation model that provides a directly reusable skeleton that can be reused in any development context. Here we describe syntactically the pattern instantiation and composition mechanism.

In order to demonstrate how our developed patterns enable reusability at different stages of development, the pattern instantiation and composition steps can be explained in Figure 9.20. We apply our proposed patterns to develop the generic communication mechanism which is needed to implement at each protocol stack layer in order to access a channel through communicating between layers. In our Event-B development, we implement this generic mechanism at the most concrete refinement level of each protocol stack layer in order to enable the final refinement model of controller to be able to interact with the simulation environment. (e.g. machines *M2* and *M5* for application and network protocols of SensorScope in Figure 9.11(b) respectively and also machines *M3* and *M5* for those protocols of RTMCS in Figure 9.11(c).) Consequently, we start to apply our patterns to implement this mechanism at the application layer as shown in Figure 9.19 as the application layer is the most abstract level of the development which can refine its generic elements to the lower layer development. This development process is divided into three steps: *pM1*, *uM2* and *pM3*. The achieved result from these three steps is a common structure instantiation for packet transmission and physical interaction that is required by our two case studies, SensorScope and RTMCS. Furthermore, we also demonstrate how this generic communication mechanism implemented in the application model can be refined to specify the specific operation for the routing model (*uM4*). Each step will be described in Sections 9.6.4.1, 9.6.4.2, 9.6.4.3 and 9.6.4.4 respectively. Here, we describe briefly each step.

- The first level of development *pM1* implements the common packet transmission. To accomplish this, patterns *PPacket*, *PSend*, *PNDBuffer*, *PDestBuffer* are considered.

- In the second level of development *uM2*, the operations for different type of packets including data and control packets are taken into account. This step refines the

Table 9.4: Overview of the pattern composition to create the generic flow of common communication

Operations	Main Corresponding Components	Provided Patterns.events	Required Patterns.events		Description
opt-1: *sensing mechanism*	SensingUnit	PsensingUnit. sensing()	-	-	The environment activates each node controller to starting sensing data. Shared event *sensing* passes the sensed data recorded in a sense buffer (*senseBuff*) to each node controller.
opt-2: *creating a packet*	Packet	PPacket. creating_pkt()	PSensingUnit. remove_sensesBuff()	PNDBuffer. record_ndBuff()	Each node creates a packet. In case of data packet, the sensed data recorded in *senseBuff* is attached in a data packet. Then, it is removed from *senseBuff*. Furthermore a created packet is recorded in a waiting buffer (*ndBuff*) for transmission.
opt-3: *sending a packet down to a channel*	Send	PSend. start_tx()	NDBuffer. remove_ndBuff()	PPacket. set_pktFwdr()	Each node controller selects a packet recorded in a *ndBuff* to initiate a packet transmission. The packet is removed from a waiting buffer after it has been transmitted.
	Send	PSend. send_down()	-	-	Shared event *send_down* indicates a packet to be transmitted down to the lower layer and a channel in MiXiM simulation.
opt-4: *sending a packet up from a channel*	Send	PSend. send_up()	-	-	Shared event *send_up* indicates a packet to be transmitted up from a channel to each receiving node. This results in the neighbour list returned from MiXiM environment to be recorded in *ctlNeighbours*.

Table 9.5: Overview of the pattern composition to create the generic flow of common communication

Operations	Main Corresponding Components	Provided Patterns.events	Required Patterns.events		Description
receiving a packet	Receive	PReceive. receive()	PSend. remove_ctlNeighbours()	-	Event *receive* indicates the receiving mechanism at each neighbour node. This results in the neighbour list recorded in *ctlNeighbours* to be moved into a receive buffer (*receiveBuff*).
opt-5: *receiving a packet for forwarding at a non-destination node*	Receive	PReceive. fwdr_receive_pkt()	PNDBuffer. record_ndBuff()	-	In case the non-destination neighbour node receives a transmitted packet. The received packet is recorded in *ndBuff* being forwarded to a destination.
opt-6: *receiving a packet for forwarding at a destination node*	Receive	PReceive. dest_receive_pkt()	PDestBuffer. record_destBuff()	-	In case the destination neighbour node receives a transmitted packet. The received packet is recorded in a destination buffer *destBuff*.
opt-7: *finishing/final transmitting a packet*	Send	PSend. finish_tx_pkt()	PNDBuffer. is_in_range_ndBuff	-	Event *finish_tx_pkt* performs the finish of each round of iterative packet forwarding transmission. This is accomplished with the condition that there are further packet in *ndBuff* to be forwarded.
	Send	PSend.final_tx_pkt	PNDBuffer. isNot_in_range_ndBuff	-	Event *final_tx_pkt* performs the final iteration of packet forwarding transmission. This is accomplished with the condition that there are no packets in *ndBuff* to be forwarded.

earlier developed model *pM1* and adds user-defined guards and actions into the refining relevant event in order to specify this specific operation.

- In the third level of development *pM3*, as the data packet creation implemented in the previous model (*uM2*) requires the sensing mechanism to complete its operation. Consequently, pattern *PSensingUnit* is considered to compose with this model.
- The fourth level of development *uM4* is a user-defined model which refines the final refinement model of application layer (*pM3*). This implements the specific operation for the routing protocol e.g. AODV for RTMCS.

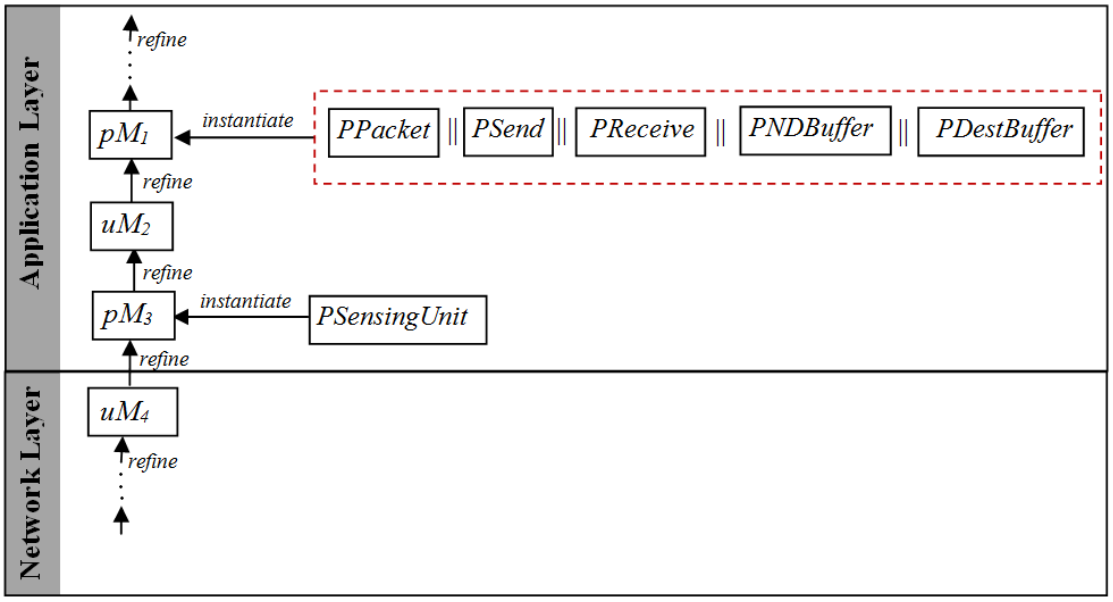


Figure 9.19: The pattern instantiation and composition steps to implement the generic communication mechanism for WSN

9.6.4.1 First Level of Development (*pM1*)

Figure 9.20 illustrates the first level of model transformation and composition of each pattern for creating the generic flow of the common communication expressed in Figure 9.18. This development process is divided into three steps: *Pattern Identification*, *Pattern Instantiation* and *Model Composition*.

Pattern Identification: at the first step, we identify the pattern that matches with the requirement. In this first level, the basic transmission communication is taken into account in which the transmitted packet is not defined into a specific type of packet e.g. data or control packet. Consequently, we include every pattern in this first level step except pattern *PSensingUnit* that is regard to the data packet as shown in Figure 9.20(a).

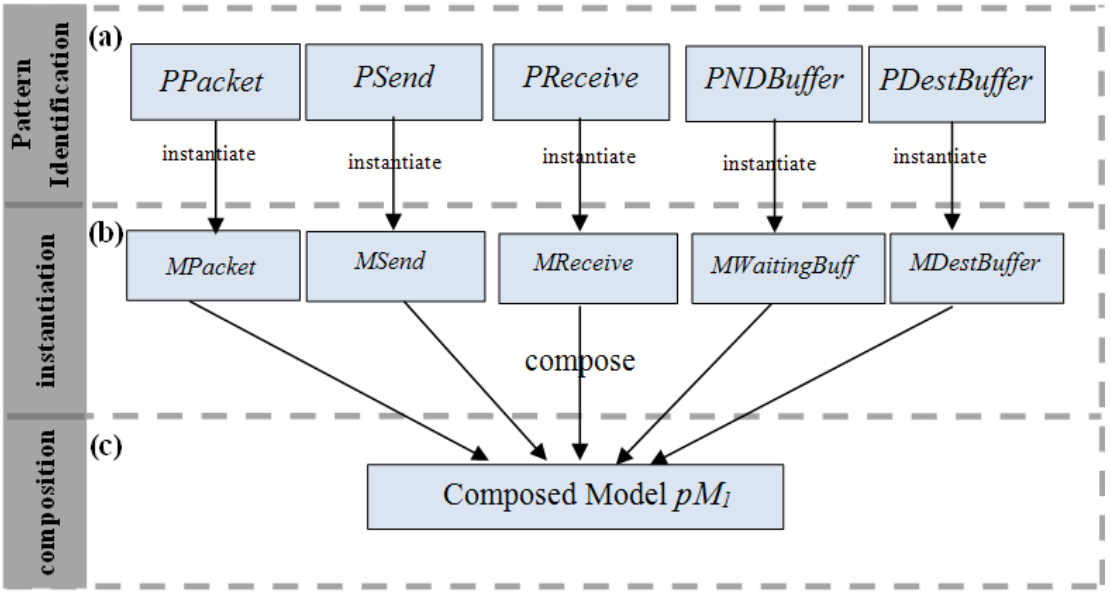


Figure 9.20: The first level of the transformation and composition producing the specification of generic communication mechanism

Pattern Instantiation: each selected pattern is instantiated into the corresponding Event-B model in instantiation step as shown in Figure 9.20(b). The generic parameter defined in the pattern is instantiated into the specific variable of target Event-B model. In this step, we assume that each pattern is consistent with the instantiated model in which there is no error during instantiation. This validity checking should be performed automatically. This is required for future work. Figure 9.21 shows an example where patterns *PPacket* and *PNDBuffer* are initialised into specific machines *mPacket* and *mWaitingBuffer* respectively. The generic model parameter $\$par\$$ is instantiated to specific elements in the target machine. For example, we replace parameter $\$record\_ndBuff\$$ with specific event name *record_waitingBuff* as well as parameter $\$ndBuff\$$ is replaced with variable name *waitingBuff* as demonstrated in Figure 9.21(b) and (c) respectively. Note that, shared events e.g. *sensing*, *send_up* and *send_down* are not allowed to change their names as they are required to have a unique name for synchronising parameters with simulation environment.

Model Composition: finally, each instantiated model is composed into the first level of target machine *pM1* by using shared-event composition techniques as expressed in Figure 9.20(c). This composed target machine *pM1* associated with this composition is expressed in Figure 9.22.

An example of composed events to create the specific operation is demonstrated in Figure 9.23. This figure demonstrates the composing event for packet creation operation (**opt-2**). The parallel composition of the generic events defined in associated patterns for defining event data packet creation is shown below:

$$pM1.creating\_pkt \triangleq MPacket.creating\_packet \parallel MWaitingBuff.record\_waitingBuff$$

```
Pattern PPacket
...
$creating_pkt$ ≙
  any x des pkt data
  where x ∈ ND ∖Dests ∧ pkt ∈ PKT
        ∧ data ∈ ℤ
        ∧ x = $initialSrcAddr$(pkt)
        ∧  des = ran({pkt} ◁
$finalDestAddr$)
        ∧ pkt  ∉ dom($pktData$)
        ...
then
     $createdPkts$:= $createdPkts$ ∪ {pkt}
     $pktData$ := $pktData$ ∪ {pkt ↦ data}
     ...
end
```

(a)

```
Pattern PNDbuffer
 ...
$record_ndBuff$ ≙
  any x pkt
  where x ∈ ND ∧ pkt ∈ PKT
        ∧ x ↦ pkt ∉   $ndBuff$
  then
     $ndBuff$ := $ndBuff$ ∪ {x ↦pkt}
  end
```

(b)

```
machine MPacket
sees cMPacket
 ...
event creating_packet
   any x des pkt data
   where
     @g1 x ∈ ND ∖Dests ∧ pkt ∈ PKT
     @g2 data ∈ ℤ
     @g3  x = initialSrc(pkt)
     @g4 des = ran({pkt} ◁ finalDest)
     @g5 pkt  ∉ dom(pktData)
   then
     @a1 createdPkts:= createdPkts ∪ {pkt}
     @a2 pktData := pktData ∪ {pkt ↦ data}
end
```

(c)

```
machine MWaitingBuff
sees cMWaitingBuff
 ...
event record_waitingBuff
   any x pkt
   where
     @g1 x ∈ ND ∧ pkt ∈ PKT
     @g2 x ↦ pkt ∉ waitingBuff
   then
     @a1 waitingBuff := waitingBuff ∪ {x ↦
pkt}
  end
```

(d)

Figure 9.21: Example of pattern instantiation: patterns (a) *PPacket*, (b) *PND-Buffer* and target instantiated machines (c) *MPacket* and (d) *MWaitingBuff* respectively

We compose event *creating_packet* of instantiated machine *MPacket* (Figure 9.21(c)) with event *record_waitingBuff* of instantiated machine *MWaitingBuff* (Figure 9.21(d)) respectively. This results in event *creating_pkt* in machine *pM1*. In this basic packet creation operation, the packet is created and recorded into the waiting buffer ready for transmission. In our corresponding patterns, event *creating_packet* in machine *MPacket* is responsible for creating a packet and assigning fields in the packet e.g. sensed data (action *@a2* in Figure 9.21(c)). Furthermore, event *record_waitingBuff* in machine *MWaitingBuff* enables the created packet to be recorded in waiting buffer as illustrated in (action *@a1* in Figure 9.21(d)). Consequently, we combine these two events together

```
COMPOSED MACHINE pM1
INCLUDES MPacket, MSend, MReceive, MWaitingBuff, MDestBuffer

EVENTS

  //opt-2: Creating a packet
   pM1.creating_pkt ≙ MPacket.creating_packet || MWaitingBuff.record_waitingBuff

  //opt-3: Sending a data packet down to a channel
   M1.start_tx ≙ MSend.start_tx || MWaitingBuff.remove_waitingBuff ||
                MPacket.set_pktFwdr
   pM1send_down ≙ MSend.send_down

  //opt-4: Sending a data packet up from a channel
   pM1.send_up ≙ MSend.send_up

  // receiving a packet
   pM1.receive ≙ MReceive.receive ||  MSend.remove_ctlNeighbours

 //opt-5: receiving a packet for forwarding at a non-destination node
  pM1.fwdr_receive_pkt ≙ MReceive. fwdr_receive_pkt || MWaitingBuff.record_waitingBuff

 //opt-6: receiving a packet for forwarding at a destination node
  pM1.dest_receive_pkt ≙ MReceive. dest_receive_pkt || MDestBuffer.record_destBuff

 //opt-7: finishing/final transmitting a packet
 pM1.finish_tx_pkt ≙ MSend.finish_tx_pkt || MWaitingBuff.is_In_Range_waitingBuff
 pM1.final_tx_pkt ≙ MSend.final_tx_pkt || MWaitingBuff.isNot_In_Range_waitingBuff
```

Figure 9.22: Composed target machine *pM1*

to create event *creating_pkt* as demonstrated in Figure 9.23 and its context defined in Figure 9.24.

```
machine pM1
sees cM1
 ...
event creating_packet
   any x des pkt data
   where
    @MPacket_creating_pkt_g1 x ∈ ND ∖ Dests ∧ pkt ∈ PKT
    @MPacket_creating_pkt_g2 data ∈ ℤ
    @MPacket_creating_pkt_g3 x = initialSrcAddr(pkt)
    @MPacket_creating_pkt_g4 des = ran({pkt} ◁ finalDestAddr)
    @MPacket_creating_pkt_g5 pkt ∉ dom(pktData)
    @MNDbuffMgt_record_waitingBuff_g1 x ↦ pkt ∉ waitingBuff
   then
    @MPacket_creating_pkt_a1 createdPkts ≔ createdPkts ∪ {pkt}
    @MPacket_creating_pkt_a2 pktData ≔ pktData ∪ {pkt ↦ data}
    @MNDbuffMgt_record_waitingBuff_a1 waitingBuff ≔ waitingBuff ∪ {x ↦ pkt}
  end
```

Figure 9.23: Composition of *PPacket* and *PWaitingBuffer* to create *creating_pkt* of machine *pM1*

```
context cM1

constants ND Dests BROADCAST initialSrc finalDest

sets PKT TYPE

axioms
 @axm1_1 ND ⊆ ℕ
 @axm1_2 finite(ND)
 @axm1_3 Dests ⊆ ND
 @axm1_4 finite(PKT)
 @axm1_5 BROADCAST = −1
 @axm1_6 initialSrc ∈ PKT → ND
 @axm1_7 finalDest ∈ PKT → ND ∪ {BROADCAST}
end
```

Figure 9.24: Context *cM1* included in machine *pM1*

9.6.4.2 Second Level of Development (*uM2*)

In the second level of model instantiation and composition, the specific type of packet is taken into account. There are two types of packet: *a data* and *control packet* included in this level. Then, we refine decomposed event *creating_pkt* achieved from the first level of development to implement the specific events for creating the two packet types, *creating_dataPkt* and *creating_controlPkt* as shown in Figure 9.25(a) with its corresponding context illustrated in Figure 9.25(b). To distinguish the difference between these two packet creation events, the following describes these two events in detail.

First of all, refining event *creating_dataPkt* is responsible for creating a data packet. The event refines abstract event *creating_pkt* and adds the user-defined guard into this refining event. In this case, guard *@User_defined_guard_g1* is added in event *creating_dataPkt* in order to define the specific type of packet that is data packet. Furthermore, this event requires the data recorded in a sense buffer to be attached in a created data packet. Consequently, we will compose this event with an instantiated event from pattern *PSensingUnit*. This will be described in the next level of development.

Considering refining event *creating_controlPkt* that represents the control packet creation, this event does not require the data recorded in a sense buffer. Thus, there are two user-defined guards added into this composed event (*creating_controlPkt*). One is *@User_defined_guard_g1* which is assigned data to be a specific unused value e.g. *CTL_VAL*. This can be any constant value defined by the developer (e.g. being 0 as defined in axiom *@axm2_4*, Figure 9.25(b)) to determine that it is not unused for a created control packet. Furthermore, another added guard is *@User_defined_guard_g2* to distinguish this event to be specific for the control packet.

```
machine uM2
refines pM1
sees uC2
...
event creating_dataPkt
refines creating_pkt
  any x des pkt data
  where
    ...
    @MPacket_creating_pkt_g5 pkt ∉ dom(pktData)
    @User_defined_guard_g1  type(pkt) = DATA
  then
    @IPacket_creating_pkt_a2 pktData ≔ pktData ∪ {pkt ↦ data}
    ...
end

event creating_controlPkt
refines creating_pkt
  any x des pkt data
  where
    ...
    @MPacket_creating_pkt_g5 pkt ∉ dom(pktData)
    @User_defined_guard_g1  data = CTL_VAL
    @User_defined_guard_g2  type(pkt) ∈ CONTROL
  then
    @IPacket_creating_pkt_a2 pktData ≔ pktData ∪ {pkt ↦ data}
    ...
end
```

(a)

```
context uC2
extends pC1

constants DATA CONTROL type

sets TYPE

axioms
  @axm2_1 DATA ∈ TYPE
  @axm2_2 CONTROL ⊆ TYPE
  @axm2_3 partition(TYPE, CONTROL, {DATA})
  @axm2_4 type ∈ PKT → TYPE
  @axm2_5 CTL_VAL = 0
end
```

(b)

Figure 9.25: Machine *uM2* refining machine *pM1* to identify the specific type of packet creation

9.6.4.3 Third Level of Development (*pM3*)

Pattern Identification: as mentioned earlier, event *creating_dataPkt* requires to attach the data recorded in a sense buffer, pattern *PSensingUnit* is selected for the third level of instantiation and composition process as illustrated in Figure 9.26(a).

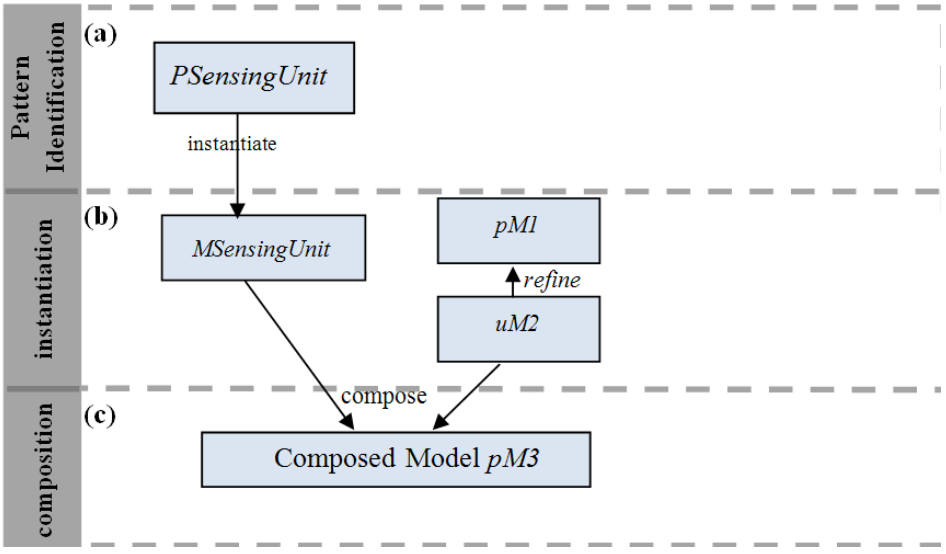


Figure 9.26: The third level of the transformation and composition producing the specification of generic communication mechanism

Pattern Instantiation: similar to the first level of development, the elements defined in pattern *PSensingUnit* are instantiated to Event-B model *MSensingUnit* in this step. Note that we replace the generic parameter provided in pattern *PSensingUnit* with the same name for the specific element in target machine *MSensingUnit*

Model Composition: this instantiated model is composed with model *uM2* to create target model *pM3* for this level as shown in Figure 9.26(c). Figure 9.27 demonstrates the composed target model *pM3*.

```
COMPOSED MACHINE pM3
INCLUDES MSensingUnit, uM2

EVENTS

  //Creating a data packet
   pM3.creating_dataPkt ≙ uM2.creating_dataPkt || MSensingUnit.remove_senseBuff

  //Creating a control packet
   pM3.creating_controlPkt ≙ uM2.creating_controlPkt
  ...
```

Figure 9.27: Composed target machine *pM3*

Considering the composed event to specify the data packet creation in composed model *pM3* (*pM3.creating_dataPkt*) in Figure 9.27, this event is created from the composition between event *creating_dataPkt* in *uM2* and event *remove_senseBuff* from instantiated machine *MSensingUnit*.

Event *remove_senseBuff* removes the sensed data from the sense buffer. This event is required by event *creating_dataPkt* to retrieve the sensed data recorded in the sense buffer as shown in guard *@ISensingUnit_remove_SenseBuff_g1*, Figure 9.28. After packet creation, such data will be removed from the sense buffer. As a result of this, we combine this instantiated event *remove_SenseBuff* from *MSensingUnit* together with event *creating_dataPkt* in model *uM2* to create event *creating_dataPkt* in composed model *pM3* as demonstrated in Figure 9.28.

9.6.4.4 Fourth Level of Development (*uM4*)

This section demonstrates how our generic pattern can facilitate the reusability. The final model of application protocol achieved in Section 9.6.4.3 can be refined to implement the specific routing protocol. This can be accomplished by adding the new event or adjusting the existing event to implement the specific functionality of routing protocol in the refinement model. For example, the new event can be added to implement the specific functionality e.g. the neighbourhood discovery, the link quality estimation and the parent selection in MintRoute and the forward and backward path setup in AODV as shown in Tables 9.1 and 9.2. However, we have discovered that these routing protocols

```
machine pM3
refines uM2
sees uC2
 ...
event creating_dataPkt
refines creating_dataPkt
any x des pkt data
 where
 @MPacket_creating_pkt_g5 pkt ∉ dom(pktData)
@MWaitingBuffMgt_record_waitingBuff_g1 x ↦ pkt ∉ waitingBuff
 @User_defined_guard_g1  type(pkt) = DATA
 @MSensingUnit_remove_SenseBuff_g1 senseBuff ≠ ∅ ∧ data ∈ senseBuff(x)
then
 @MPacket_creating_pkt_a2 pktData ≔ pktData ∪ {pkt ↦ data}
 @MWaitingBuffMgt_record_ waitingBuff _a1 waitingBuff ≔ waitingBuff ∪ {x ↦
pkt}
@MSensingUnit_remove_SenseBuff_a1 senseBuff ≔ senseBuff \ {x ↦ pkt}
 ...
end
```

Figure 9.28: The result after the composition between corresponding event *removeSenseBuff* in instantiated machine *MSensing* that are composed with event *creating_dataPkt* in model *uM2*

require the basic transmission and communication e.g. the control packet flooding to process their algorithms and maintain the network topology (Akkaya and Younis, 2005). As a result of this, the event corresponding to this mechanism e.g. *creating_controlPkt* can be refined and adjusted as shown in Figure 9.29. As can be seen in this figure, we refine model *pM3* to implement the model *uM4* for the protocol AODV. As this protocol floods a route request packet (RREQ) and unicasts a route reply packet (RREP) in order to form the backward and forward path respectively, we partition set CONTROL and adjust guard to identify this specific packet. Furthermore, we refine event *creating_controlPkt* to implement the concrete event for this purpose. For example, in Figure 9.29(b), this event is refined to implement event *creating_RReqPkt* and adjust guard *@User_defined_guard_g4* to identify the route request flooding operation. We have also discovered, by comparing this implementation to the MintRoute development, this can be developed in the same way to identify BEACON and ROUTE packet operation.

9.6.4.5 Example of Abstract and Realising Patterns

In the earlier section, we have explained abstract and realising patterns in which the realising pattern can refine the elements defined in the abstract pattern in order to define the specific operation at implementation level. Although these type of patterns are a work in progress, a possible example can be illustrated based on our experiences of SensorScope development. Here, we demonstrate this example in the following.

```
context uC3
extends uC2

constants RREQ RREP
axioms
 @axm3_1 partition(CONTROL, {RREQ}, {RREP})
end
```

(a)

```
machine uM4
refines pM3
sees uC4
 ...

event creating_RReqPkt
refines creating_controlPkt
  any xdes pkt data
  where
    ...
    @User_defined_guard_g4  type(pkt) = RREQ
  then
    @IPacket_creating_pkt_a2 pktData ≔ pktData ∪ {pkt ↦ data}
    ...
end
```

(b)

Figure 9.29: Example of the refinement (a) context and (b) machine to implement the specific operation for AODV protocol

In our SensorScope development in Event-B, the abstract model for the basic transmission and communication was implemented by abstracting the protocol layer. This enabled each node to transmit its packet abstractly to its destination through the wireless medium (*WiMedium*) as shown in Figure 9.30(a). Then, we refined this model by introducing the protocol stack into the concrete model. This implemented the sending up and down operations of each layer in our controller model so that they could be integrated in to the lower layer and a channel in MiXiM environment as shown in Figure 9.30(b).

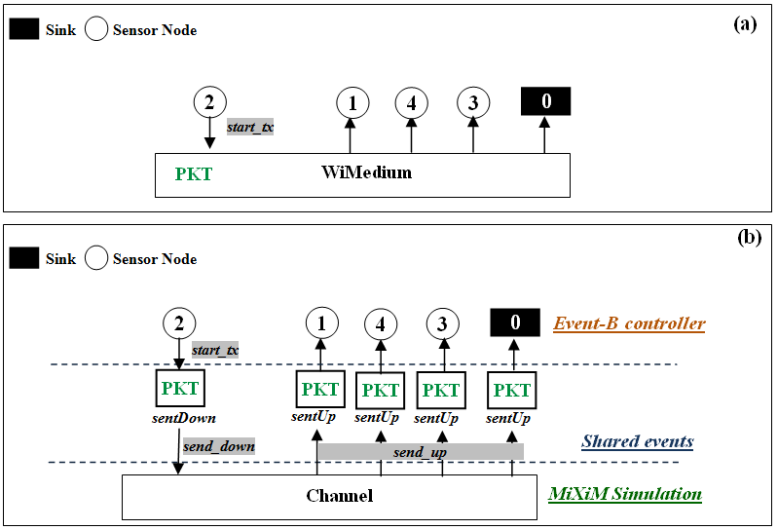


Figure 9.30: An example of (a) abstract and (b) realising patterns for the basic transmission and communication

As a result of this, we can define the abstract pattern of pattern *PSend* as shown in Figure 9.31(a) by introducing variable $\$WiMedium\$$. This variable is refined down into variables $\$sentUp\$$ and $\$sentDown\$$ in the realising pattern $\$PSend\$$ as shown in Figure 9.31(b). Gluing invariant *partition*($\$WiMedium\$$, $\$sentUp\$$, $\$sentDown\$$) was introduced in order to strengthen the consistency of this data refinement in realising

pattern *ISend*. This also affects the events in the abstract pattern e.g $\$start\_tx\$$ that maintain variable $\$WiMedium\$$ to replace this abstract variable with these concrete variables $\$sentUp\$$ and $\$sentDown\$$ in the realising pattern.

```
Pattern PSend0
   include Context_PSend0
   variable  $WiMedium$
      invariant  $WiMedium$  ∈ ND ↔PKT
         action  $WiMedium$  ≔ ∅
   end

$start_tx$  ≙  any  x pkt
            where x ∈ ND ∧ pkt ∈ PKT
                  ∧ x ↦ pkt ∉ $WiMedium$
            then  $WiMedium$  ≔ $WiMedium$  ∪  {x ↦
pkt}
            end
...
```

(a)

```
Pattern PSend
Realizes PSend0
   include Context_ISend
   variable $sentDown$ $SentUp$
      invariant $sentDown$  ∈ ND ↔PKT
            ∧ $SentUp$  ∈ ND ↔PKT
         action $sentDown$≔ ∅ ||
               $SentUp$≔ ∅
   invariant partition(WiMedium, $SentUp$, $sentDown$
   end

$start_tx$  ≙  any            x pkt
            where x ∈ ND ∧ pkt ∈ PKT
                  ∧ x ↦ pkt ∉ $sentDown$
            then  $sentDown$ ≔ $sentDown$  ∪  {x ↦ pkt}
                  $SentUp$≔ ∅ ||
             end
...
```

(b)

Figure 9.31: An example of abstract and realising patterns (a) abstract pattern *PSend0* and (b) realising pattern *PSend*

9.7 Conclusion

The main motivation of this chapter is to provide usability and reusability in our co-simulation framework. This also demonstrates with F^S *Co-simulation* development that starts with the protocol modelling in F-style to C-style through S-style development. The goal is achieved by measuring the level of reusability from our achieved co-model, SensorScope, to another different case study, RTMCS. The results have shown that our co-simulation framework supports the usability and reusability for the following three aspects: Event-B controller development, master and MiXiM environment implementation.

First of all, based on the result of measurement of Event-B model reuse between SensorScope and RTMCS, we have discovered that the application protocol in SensorScope enables a high reuse rate compared to the routing protocol. This is because the application protocol contains the generic pattern e.g. sensing, packet transmission and physical interaction mechanisms shared between these two case studies. However, the routing protocol of these two case studies is completely different in which they contain the specific functionality to perform the specific routing algorithm.

Furthermore, we have discovered the refinement patterns of controller development in which the functionality of the protocols layered on the top of the packet transmission and physical interaction mechanism at each layer of protocol stack. These mechanisms

are generic and therefore we establish generic patterns based on these mechanisms. We can then apply these patterns to explore some reusability. The result of the application of this generic pattern has demonstrated the level of reusability. We can totally reuse and combine our proposed generic patterns to create the basic packet transmission mechanism in a target Event-B model at the application layer through the refinement development chain. The common reusable instantiation model, a result of this pattern instantiation and composition, gives a directly reusable skeleton that can be reused in any development context. The specific functionality of specific application can be layered on the top on this generic model or it can refine to create a concrete model for the routing protocol by layering the specific routing protocol functionality as we have demonstrated in Figure 9.6 for SensorScope and RTMCS development. For the proof of correctness purpose, we have not considered the support for proof reuse and proof requirements. An analogous tool support to theory plug-in that allows the structure of our generic patterns to be instantiated and composed to create more specific model. This includes the invariant instantiation in the instantiated and composed component that enables the corresponding generic proof obligations to be reused.

Finally, the analysis of the structure of master and MiXiM environment has shown us that it supports the reusability. The interface classes or modules are implemented in these components. For example, interface `EventBCtl` implemented in a master encodes the general interaction and communication mechanism between an Event-B controller via shared events *sensing*, *send_down* and *send_up* and MiXiM simulation environment. Furthermore, Interfaces `FMInterface`, `SimManager`, `SimpleApp` and `SimpleNet` implemented in the MiXiM environment provide the standard external interaction between the Event-B controller and MiXiM environment models and internal communication between two upper layers as demonstrated in Chapter 7.

Chapter 10

Conclusions

This thesis provides a hybrid approach between proof-based formal engineering with simulation-based development methodologies to strengthen the current practice of WSN development. This approach aims to provide the methodology and technique that increases the quality of WSN development including high-level abstraction, separation of concerns and powerful V&V techniques. This thesis has demonstrated the infrastructure for co-simulation between Event-B controller and MiXiM environment simulation engines. This chapter summaries our achievements and directions for future work.

10.1 Main Contribution

The main contribution of this thesis is a formal co-simulation approach (*FoCoSim-WSN*) to facilitate good SE methodologies and techniques for WSN development. To achieve this goal, we developed a prototype framework by integrating the strength between proof-based formal development with simulation-based testing techniques. The result of this development enables an iterative interworking development framework for (F)ormal, (S)imulation and (C)osimulation. This provides the system engineers with full confidence in the improvement of the development and high quality in terms of correctness, reliability and performance of their deployed application.

This research has achieved five sub-contributions: C.1 the integration between proof-based formal verification and simulation-based testing techniques, C.2 methodological flexibility in co-modelling development, C.3 formal co-simulation interfaces contributing to reusability, C.4 model-based trace testing (MBTT) technique and C.5 usability and reusability in co-model development.

10.1.1 The integration between proof-based formal verification and simulation-based testing techniques

One of the major aims of our FoCoSim-WSN development is to integrate the benefit of proof-based formal verification and simulation-based testing techniques targeting Event-B and MiXiM respectively. The results of both developments achieved from Chapters 5 and 6 have shown that they can compensate for the weakness of each other. Safety invariants for safety properties of protocol algorithms including basic communication and routing protocol (e.g. loop problems) are guaranteed by use of Event-B proof tools, whereas the performance of such an algorithm e.g. the network load distribution and the network latency is analysed and evaluated by using WSN simulation.

Furthermore, we have demonstrated the benefit of refinement and decomposition provided by Event-B development to tackle the complexity of protocol algorithm development in simulation discovered in Chapter 5. By using the refinement technique, the developed single simulation model for each protocol stack is separated and implemented into multiple layers as demonstrated in Section 6.3 of Chapter 6. Furthermore, by applying shared-event decomposition technique, the complexity of large system development is managed by breaking a single machine into sub-machines. We used this decomposition technique to decompose the node controller model from the physical environment model. Shared events are identified in these sub-models to interact with each other by exchanging messages as described in Section 6.7. This enables the high-level abstraction and separation of concerns to WSN development.

10.1.2 Methodological flexibility in co-modelling development

After the experiment of the co-simulation case study described in Section 7.6, we have discovered that our co-simulation prototype provides flexible modes of co-model development. The developed models from both F-style formal-based and S- style simulation-based developments can be reused. This creates the iterative and incremental co-model development for each layer protocol in which Event-B controller, representing the upper layer model, can communicate with MiXiM physical environment models. This results in multiple levels of co-simulation abstraction that distributes the functionality of WSN systems including protocol algorithms between components as demonstrated in Section 7.6. Our SensorScope case study development was demonstrated with three levels of abstraction: no partition of development (S-level), the abstraction level of application protocol ($S^{Fo.2}$ for SensorApp) and the abstraction level of network protocol ($S^{Fo.5}$ for MintRoute).

Futhremore, this framework provides the flexibility and utility integration of F-, S- and C-style development. This enables the model to be developed over these three flexible modes of working. We have proposed and demonstrated two strategies of interworking

evolution: *from-S-through-F-to-C*($S^F Co-simulation$) (Chapter 7) and *from-F-through-S-to-C* ($F^S Co-simulation$) (Chapter 9) which enables system engineers to work either F- or S-style development before combining and co-simulating the separated controller and environment models in co-simulation framework.

10.1.3 Formal co-simulation interfaces contributing to reusability

We developed independent and reusable interfaces for our proposed co-simulation framework. These interfaces are divided into two categories: *master interfaces* and *MiXiM interfaces*. On the one hand, the master interfaces aim to access and schedule events in the Event-B model. For example, controller interface class `EventBCtl` expressed in Chapter 7 and Chapter 9, can be implemented later to the specific application e.g. SensorScope and RTMCS in order to access and schedule event traces in Event-B controller model. The input/output parameters also are created and passed to physical simulation environment by this interface. Furthermore, these parameters are encoded into standard XML format.

On the other hand, we develop MiXiM interfaces to pass/extract input/output parameters to/from physical simulation environment. These interfaces also exchange input/output parameters between controller formal and physical simulation environment via a socket API interface. As described in Chapter 7, two interfaces are developed *FMInterface* and *SimManager*. The former is a front-end interface for synchronising parameters between its synchronization event and the shared-event in the Event-B model. The latter is the back-end interface which pass the parameters down to/up from the channel.

10.1.4 Model-based trace testing (MBTT) technique

As can be seen clearly from the result achieved in Chapter 8, the application of MBTT technique can examine the test coverage for formal controller model validation in WSN co-simulation. Various test scenarios including functional, failure and recovery test scenarios can be derived from a formal Event-B model which is used as the model under test (MUT). These test scenarios can be derived to create testing experiments and configured in the configuration file for MiXiM, test driver in co-simulation framework. We discovered that failure test scenarios ("killer traces") were enabled to detect and debug absent or erroneous constraints and events. We believe that the validation coverage provided by this MBTT approach can strengthen the V&V including proof-based verification and simulation-based testing techniques provided by FoCoSim-WSN framework to enable early detection of defects in WSN systems before the deployment.

10.1.5 Usability and reusability and in co-model development

Further exploration of our developed FoCoSim-WSN framework in terms of reusability and usability for potential development was also conducted. This is demonstrated by the second case study, RTMCS. We reuse the physical interaction and communication mechanisms via shared events from our previous implementation. This creates the generic pattern in which the specific operation of protocol algorithms can be layered on the top as demonstrated in Chapter 9.

10.2 Future Work

The results shown in this thesis show that the Formal Co-Simulation Framework for WSN development (FoCoSim-WSN) can strengthen current SE practice of WSN development. However, there are some challenges that can improve this developed framework. Here are the possible research directions.

Further formal verification in Event-B. In our Event-B development and verification, the safety invariants expressing the loop free and reachability property in the route tree from source node to a sink give a guarantee of basic reliable end-to-end communication (cf. Section 6.3.7). To improve the confidence of this verification, we could consider the liveness property to guarantee that "*a transmitted packet will eventually reach a sink/destination*". We have discovered that non-divergence condition proposed in the study of Butler (1992) could be applied to prove this liveness property. In this work (Butler, 1992), the hop length, a number of immediate nodes along the route between source and a sink, is introduced as a variant and prove that it decreases when the packet is transmitted from one node to the next immediate node along this route. This hop length could be calculated by our implemented transitive closure (*tcl*) as it identifies the reachable path from source to destination node. To guarantee this property, in our machine *M5* (cf. Section 6.3.7), event *receive_dataPkt* (refinement event *receive_pkt* in machine *M2*) could be defined as a convergent event that must decrease this variant in order to prove that it cannot execute forever. When this variant decreases to be 0, the event *sink_receive_pk* is enabled. This means that a transmitted packet has reached a sink. As a result, this could confirm this reachability in our Event-B model.

Lower layer protocol co-modelling and co-simulation. As we have already mentioned, this research focuses on the experiment at the two upper layers of communication protocol stack, our developed formal co-simulation framework currently supports the co-model and co-simulation for application and network layers e.g. SensorApp and MintRoute as illustrated in Chapter 7. In order to improve this framework efficiency, the adjustment of FoCoSim-WSN to support the co-simulation development for lower layers e.g. MAC is an interesting future direction. This can be accomplished in the

same way as we have done for the two upper layers. $S^F Co - simulation\ development$ can be used as a guideline for this adjustment as already explained in Section 7.9.2.

Multi-cosimulation development. Another interesting direction would be to extend our co-simulation framework to support multi-cosimulation. There are two possible approaches to accomplish this. One is the use of the socket interface provided by our co-simulation framework to co-simulate with the other simulation framework. One of the simulation frameworks that we are interested to integrate is the energy harvesting model implemented in HarvWSNet framework (Didioui et al., 2013). This model is implemented with MATLAB and integrated loosely with the protocol stack model provided by WSNet[1] simulation (Fraboulet et al., 2007). TCP socket interfaces are provided at both sides in order to create a co-simulation environment for parameter exchange between these two models. As a result of this, to strengthen the simple battery model provided in MiXiM environment, it can be possible to separate energy harvesting model together with its socket interface and integrate with our co-simulation framework. Another possible approach for multi-cosimulation development is to apply FMI technologies to integrate our co-simulation with other simulation frameworks. As there are several simulations supporting FMI standards such as Simulink[2], Modelica[3] and also our Master groovy implemented on ProB (ADVANCE, 2013), this can benefit our master algorithm to be integrated with other simulations via this FMI standard.

Automatic MBTT framework/tool support. As already discussed in Section 8.4 that our MBTT testing framework proposed in this thesis requires mapping between abstract and concrete test scenarios to be accomplished automatically. Thus, another direction of further research is to develop an automatic translation between these two models. An open research issue is the potential framework/tool support for MBTT to provide the automatic MBTT process to improve model validation.

Co-simulation framework at node level. Another possible research issue is the co-simulation experiment at node level as discussed in Figure 1.1, Chapter 1. The verified network node can be refined further down with requirements for the specific dependent platform at node level. This can be deployed into the real hardware (e.g. motes) that can be simulated with simulation/emulation at node level. One possible approach is the application of Event-B code generation for co-simulation using FMI developed by (Edmunds, 2013, 2014)[4] to generate specific C programming code with FMI from Event-B specification models. As FMI has been recently adopted into TinyOs and AVRORA[5] emulator (Titzer and Palsberg, 2005; Titzer et al., 2005b) as proposed by Wang and Baras (2013), we believe that this generated formal code can co-simulate

[1]see WSNet - http://wsnet.gforge.inria.fr/
[2]see Simulink (Simulation and Model-Based Design) - http://uk.mathworks.com/products/simulink/
[3]see Modelica - https://modelica.org
[4]see Code Generation for Co-simulation using FMI - http://wiki.event-b.org/index.php/Code_Generation_Activity
[5]see Avrora - http://compilers.cs.ucla.edu/avrora/

with AVRORA emulator. Another possible research issue is the extension of MBTT for validating generated node controller code as SUT (System Under Test).

Appendix A

The Event-B Model of the SensorScope Case Study

A.1 Abstract specification: basic communication protocol

A.1.1 Context: C0

CONTEXT C0

SETS

PKT

CONSTANTS

$Sink$

ND

$BROADCAST$

$initialSrcAddr$

AXIOMS

axm0_1 : $ND \subseteq \mathbb{N}$ //set of nodes in the network

axm0_2 : $finite(ND)$ //nodes should be finite

axm0_3 : $Sink \in ND$ //a sink node should be a member in set of nodes

axm0_4 : $initialSrcAddr \in PKT \rightarrow ND$ //a packet structure: source of packet

axm0_5 : $BROADCAST = -1$ //broadcast code

END

A.1.2 Machine: M0

MACHINE M0

SEES C0

VARIABLES

$middleware$ //middleware

$sinkBuff$ //a sink buffer

$xmittedPkts$ //transmitted packets

$lostPkts$ //lost packets

INVARIANTS

int0_1 : $xmittedPkts \subseteq PKT$

inv0_2 : $middleware \subseteq PKT$

inv0_3 : $sinkBuff \subseteq PKT$

inv0_4 : $lostPkts \subseteq PKT$

EVENTS

Initialisation

begin

int0_1 : $xmittedPkts := \varnothing$

int0_2 : $middleware := \varnothing$

int0_3 : $sinkBuff := \varnothing$

int0_4 : $lostPkts := \varnothing$

end

Event $create\_pkt \mathrel{\widehat{=}}$

//nodes create a packet

any

s

pkt

where

g1 : $pkt \in PKT$

g2 : $pkt \notin xmittedPkts$

g3 : $pkt \notin middleware$

g4 : $s = initialSrcAddr(pkt)$

then

a1 : $xmittedPkts := xmittedPkts \cup \{pkt\}$

a2 : $middleware := middleware \cup \{pkt\}$

end

Event $sink\_recv\_pkt \mathrel{\widehat{=}}$

//sink receives a packet

any

s

pkt

where

g1 : $s = initialSrcAddr(pkt)$

g2 : $s \neq Sink$

g3 : $pkt \in middleware \setminus sinkBuff$

then

a1 : $sinkBuff := sinkBuff \cup \{pkt\}$

end

Event *final_tx_pkt* $\widehat{=}$

//the transmitted packet is removed from middleware indicating completion of transmission mechanism

any

pkt

where

g1 : $pkt \in PKT$

g2 : $pkt \in middleware$

then

a1 : $middleware := middleware \setminus \{pkt\}$

end

Event *net_losing* $\widehat{=}$

//network loses packet

any

pkt

where

g1 : $pkt \in middleware \setminus lostPkts$

then

a1 : $middleware := middleware \setminus \{pkt\}$

a2 : $lostPkts := lostPkts \cup \{pkt\}$

end

END

A.2 First refinement: neighbour nodes

A.2.1 Context:C1

CONTEXT C1

EXTENDS C0

SETS

$TYPE$

CONSTANTS

$DATA$

$CONTROL$

$type$

$FAILED\_XMIT$

AXIOMS

axm1_1 : $DATA \in TYPE$

axm1_2 : $CONTROL \subseteq TYPE$

axm1_3 : $partition(TYPE, CONTROL, \{DATA\})$

//two packet types: control and data packets

axm1_4 : $type \in PKT \rightarrow TYPE$ //type of packets

axm1_5 : $FAILED\_XMIT \in \mathbb{Z}$ //constant for failed transmission

END

A.2.2 Machine:M1

MACHINE M1

REFINES M0

SEES C1

VARIABLES

$sinkBuff$ //a sink buffer

$ndBuff$ //nodes' waiting buffer

$WiMedium$ //wireless medium

$sensingNDs$ //sensing nodes

$sensedPkts$ //sensed packets

$floodedPkts$ //flooded packets

$floodTbl$ //historical received packets used for detect duplicated packets

$ctlNeighbours$ //neighbour lists

$recLostPkts$ //recorded lost packet at each node

$floodFlg$ //flood flag

INVARIANTS

inv1_1 : $ndBuff \in ND \leftrightarrow PKT$

inv1_2 : $WiMedium \in ND \leftrightarrow PKT$

inv1_3 : $ctlNeighbours \in PKT \leftrightarrow (ND \cup \{FAILED\_XMIT\})$

inv1_4 : $sensingNDs \subseteq ND \setminus \{Sink\}$

inv1_5 : $sensedPkts \subseteq PKT$

inv1_6 : $floodedPkts \subseteq PKT$

inv1_7 : $floodTbl \in ND \rightarrow \mathbb{P}(PKT)$

inv1_8 : $floodFlg \in ND \rightarrow BOOL$

inv1_9 : $recLostPkts \in ND \leftrightarrow PKT$

glu1_1 : $ran(ndBuff \cup WiMedium) = middleware$

glu1_2 : $sensedPkts \cup floodedPkts = xmittedPkts$

glu1_3 : $ran(recLostPkts) = lostPkts$

saf1_1 : $ndBuff \cap WiMedium = \varnothing$

saf1_2 : $(ndBuff \cap WiMedium) \cap recLostPkts = \varnothing$

EVENTS

Initialisation

begin

int0_3 : $sinkBuff := \varnothing$

int1_1 : $ndBuff := \varnothing$

int1_2 : $WiMedium := \varnothing$

int1_3 : $sensingNDs := \varnothing$

int1_4 : $sensedPkts := \varnothing$

int1_6 : $floodedPkts := \varnothing$

int1_7 : $floodTbl := ND \times \{\varnothing\}$

int1_8 : $ctlNeighbours := \varnothing$

int1_12 : $floodFlg := ND \times \{FALSE\}$

inv1_20 : $recLostPkts := \varnothing$

end

Event *create_dataPkt* $\widehat{=}$

//creating a data packet

refines *create_pkt*

any

s

pkt

where

g1 : $pkt \in PKT \wedge s \in ND \setminus \{Sink\}$

g2 : $pkt \notin ran(ndBuff \cup WiMedium)$

g3 : $s \notin sensingNDs \wedge pkt \notin sensedPkts \cup floodedPkts$ //has not been created yet

g4 : $s = initialSrcAddr(pkt)$

g5 : $s \in dom(floodTbl) \wedge pkt \notin floodTbl(s)$

g6 : $type(pkt) = DATA$

then

a1 : $sensingNDs := sensingNDs \cup \{s\}$

a2 : $sensedPkts := sensedPkts \cup \{pkt\}$
a3 : $ndBuff := ndBuff \cup \{s \mapsto pkt\}$
a4 : $floodTbl(s) := floodTbl(s) \cup \{pkt\}$
end

Event *clear_sensingNDs* $\widehat{=}$
//clearing a sensed node
any
x
where
g1 : $sensingNDs = \{x\}$
then
a1 : $sensingNDs := \varnothing$
end

Event *create_controlPkt* $\widehat{=}$
//creating a control packet
refines *create_pkt*
any
s
pkt
where
g1 : $pkt \in PKT$
g2 : $pkt \notin ran(ndBuff \cup WiMedium)$
g3 : $pkt \notin sensedPkts \cup floodedPkts$ //has not been created yet
g4 : $s = initialSrcAddr(pkt)$
g5 : $s \in dom(floodTbl) \wedge pkt \notin floodTbl(s)$
g6 : $type(pkt) \in CONTROL$
g7 : $floodFlg(s) = TRUE$
then
a1 : $floodedPkts := floodedPkts \cup \{pkt\}$
a2 : $ndBuff := ndBuff \cup \{s \mapsto pkt\}$
a3 : $floodTbl(s) := floodTbl(s) \cup \{pkt\}$
end

Event *start_tx* $\widehat{=}$
//starting transmitting a packet to wireless medium
any
x
pkt
where
g1 : $x \mapsto pkt \in ndBuff$ //packet in waiting buffer
g2 : $x \mapsto pkt \notin WiMedium$ //but has not been transmitted to wireless medium

then

a1 : $ndBuff := ndBuff \setminus \{x \mapsto pkt\}$
a2 : $WiMedium := WiMedium \cup \{x \mapsto pkt\}$

end

Event *indicate_neighbours* $\widehat{=}$

//channel indicates the neighbours of a forwarder

any

pkt
$nbrs$
f

where

g1 : $pkt \in ran(WiMedium)$
g2 : $f \mapsto pkt \in WiMedium$
g3 : $nbrs \in \{n | n \in (\mathbb{P}(ND)) \cup \{\{FAILED\_XMIT\}\}\}$
g4 : $nbrs \neq \varnothing$
g5 : $pkt \notin dom(ctlNeighbours)$

then

a1 : $ctlNeighbours := ctlNeighbours \cup (\{pkt\} \times nbrs)$

end

Event *receive_pkt* $\widehat{=}$

//each neighbour node recieves a transmitted packet

any

s
nb
pkt

where

g1 : $pkt \in ran(WiMedium)$
g2 : $pkt \mapsto nb \in ctlNeighbours$
g3 : $nb \mapsto pkt \notin ndBuff$
g4 : $nb \mapsto pkt \notin WiMedium$
g5 : $s = initialSrcAddr(pkt) \wedge s \neq nb$
g6 : $nb \neq Sink$
g7 : $nb \in dom(floodTbl) \wedge pkt \notin floodTbl(nb)$ //is not a duplicated packet

then

a1 : $ndBuff := ndBuff \cup \{nb \mapsto pkt\}$
a2 : $ctlNeighbours := ctlNeighbours \setminus \{pkt \mapsto nb\}$
a3 : $floodTbl(nb) := floodTbl(nb) \cup \{pkt\}$

end

Event *receive_dup_pkt* $\widehat{=}$

//each neighbour node discards a duplicated packet

any

nb

pkt

where

g1 : $pkt \in dom(ctlNeighbours)$

g2 : $pkt \mapsto nb \in ctlNeighbours$

g3 : $nb \in dom(floodTbl) \wedge pkt \in floodTbl(nb)$ //is a duplicated packet

then

a1 : $ctlNeighbours := ctlNeighbours \setminus \{pkt \mapsto nb\}$

end

Event *sink_recv_pkt* $\widehat{=}$

//sink receives a packet

refines *sink_recv_pkt*

any

s

pkt

nb

where

g1 : $s = initialSrcAddr(pkt)$

g2 : $s \neq Sink$

g3 : $pkt \in ran(WiMedium) \setminus sinkBuff$

g4 : $nb = Sink$

g5 : $pkt \mapsto nb \in ctlNeighbours$

g6 : $nb \in dom(floodTbl) \wedge pkt \notin floodTbl(nb)$ //is not a duplicated packet

then

a1 : $sinkBuff := sinkBuff \cup \{pkt\}$

a2 : $ctlNeighbours := ctlNeighbours \setminus \{pkt \mapsto nb\}$

a3 : $floodTbl(nb) := floodTbl(nb) \cup \{pkt\}$

end

Event *finish_tx_pkt* $\widehat{=}$ //finishing packet transmission

any

f

pkt

where

g1 : $\{pkt\} \lhd ctlNeighbours = \varnothing$

g2 : $pkt \in ran(WiMedium) \wedge pkt \in ran(ndBuff)$

//there are some packets being forwarded

g3 : $f \in dom(WiMedium) \wedge f \mapsto pkt \in WiMedium$

then

a1 : $WiMedium := WiMedium \setminus \{f \mapsto pkt\}$

end

Event *final_tx_pkt* $\widehat{=}$

//the transmitted packet is removed from middleware indicating completion of transmission mechanism

refines *final_tx_pkt*

any

pkt

where

g1 : $\{pkt\} \lhd ctlNeighbours = \varnothing$

g2 : $pkt \in ran(WiMedium) \wedge pkt \notin ran(ndBuff)$

then

a1 : $WiMedium := WiMedium \rhd \{pkt\}$

end

Event *lose_pkt* $\widehat{=}$

//each node records a lost packet

refines *net_losing*

any

f

pkt

where

g1 : $pkt \in ran(WiMedium) \setminus ran(recLostPkts)$

g2 : $pkt \notin ran(ndBuff)$

g3 : $f \mapsto pkt \in WiMedium$

g4 : $f \mapsto pkt \notin recLostPkts$

g5 : $ran(\{pkt\} \lhd ctlNeighbours) = \{FAILED\_XMIT\}$

then

a1 : $WiMedium := WiMedium \rhd \{pkt\}$

a2 : $recLostPkts := recLostPkts \cup \{f \mapsto pkt\}$

end

END

A.3 Second refinement: network topology, sensing mechanism, physical environment and channel mechanism

A.3.1 Context:C2

CONTEXT C2

EXTENDS C1

CONSTANTS

WSN

$wsnTopology$

$randomFn$

$minR$

$maxR$

AXIOMS

axm2_1 : $WSN = ND \leftrightarrow ND$ //wireless sensor network structure

axm2_2 : $wsnTopology \in WSN$ //wireless sensor network topology

axm2_3 : $randomFn = minR \mathinner{..} maxR$
//basic random function for environment

END

A.3.2 Machine:M2

MACHINE M2

REFINES M1

SEES C2

VARIABLES

$sinkBuff$ //(ctl) a sink buffer

$ndBuff$ //(ctl) nodes' waiting buffer

$sensingNDs$ //(ctl) sensing nodes

$sensedPkts$ //(ctl) sensed packets

$senseBuff$ //(ctl) sense buffer

$floodedPkts$ //(ctl) flooded packets

$floodTbl$ //(ctl)historical received packets used for detect duplicated packets

$ctlNeighbours$ //(ctl) neighbour lists

$recLostPkts$ //(ctl) recorded lost packet at each node

$envSensedFlg$ //(env) sensing period

$envData$ //(env) environmental/conditional data

$ctlSensedFlg$ //(ctl) sensing period

$sentDown$ //(ctl) transmitted packets down to channel

$sentUp$ //(ctl) transmitted packets up from channel

$channel$ //(env) channel

$envNeighbours$ //(env) neighbour list

$wsnLinks$ //(env) dynamic WSN links

$floodFlg$ //(ctl) flooded flag

$dataSeqNo$ //(ctl) local nodes' current data sequence number at application layer

$floodSeqNo$ //(ctl) local nodes' current broadcast sequence number at application layer

$nbHops$ //(ctl) neighbour's hop count

$crashedLinks$ //(env) crashed WSN Links

$pktSrc$ //(ctl) src address attached in transmitted packet

$pktSeqNo$ //(ctl) sequence number attached in transmitted packet

$pktFwdr$ //(ctl) fowarder attached in transmitted packet

$pktData$ //(ctl) data attached in transmitted packet

$pktNbHops$ //(ctl) hop count attached in transmitted packet

$vPktSeqNo$ //(env) virtual packet at channel

$vPktSrc$ //(env) virtual packet at channel

$vPktFwdr$ //(env) virtual packet at channel

$vPktData$ //(env) virtual packet at channel

$vPktNbHops$ //(env) virtual packet at channel

INVARIANTS

inv2_1 : $senseBuff \in ND \setminus \{Sink\} \rightarrow \mathbb{P}(\mathbb{Z})$

inv2_2 : $sentUp \in ND \leftrightarrow PKT$

inv2_3 : $sentDown \in ND \leftrightarrow PKT$

inv2_4 : $ctlSensedFlg \in ND \setminus \{Sink\} \rightarrow BOOL$

inv2_5 : $dataSeqNo \in ND \rightarrow \mathbb{N}$

inv2_6 : $floodSeqNo \in ND \rightarrow \mathbb{N}$

inv2_7 : $nbHops \in ND \leftrightarrow (PKT \nrightarrow \mathbb{Z})$

inv2_8 : $pktSeqNo \in PKT \nrightarrow \mathbb{N}$

inv2_9 : $pktSrc \in PKT \nrightarrow ND$

inv2_10 : $pktFwdr \in PKT \nrightarrow ND$

inv2_11 : $pktData \in PKT \nrightarrow \mathbb{Z}$

inv2_12 : $pktNbHops \in PKT \nrightarrow \mathbb{Z}$

env2_1 : $wsnLinks \in WSN$

env2_2 : $(ND \lhd id) \cap wsnLinks = \varnothing$

env2_3 : $channel \in ND \leftrightarrow PKT$

env2_4 : $envNeighbours \in ran(channel) \leftrightarrow (ND \cup \{FAILED\_XMIT\})$

env2_5 : $envSensedFlg \in ND \setminus \{Sink\} \rightarrow BOOL$

env2_6 : $envData \in ND \setminus \{Sink\} \rightarrow \mathbb{Z}$

env2_7 : $vPktNbHops \in ran(channel) \nrightarrow \mathbb{Z}$

env2_8 : $vPktSrc \in ran(channel) \nrightarrow ND$

env2_9 : $vPktSeqNo \in ran(channel) \nrightarrow \mathbb{N}$

env2_10 : $vPktFwdr \in ran(channel) \rightarrow ND$

env2_11 : $vPktData \in ran(channel) \rightarrow \mathbb{Z}$

env2_12 : $crashedLinks \in WSN$

glu2_1 : $partition(WiMedium, sentDown, sentUp)$

EVENTS

Initialisation

begin

int0_3 : $sinkBuff := \varnothing$

int1_1 : $ndBuff := \varnothing$

int1_3 : $sensingNDs := \varnothing$

int1_4 : $sensedPkts := \varnothing$

int1_5 : $senseBuff := (ND \setminus \{Sink\}) \times \{\varnothing\}$

int1_6 : $floodedPkts := \varnothing$

int1_7 : $floodTbl := ND \times \{\varnothing\}$

int1_8 : $ctlNeighbours := \varnothing$

int1_9 : $envSensedFlg := (ND \setminus \{Sink\}) \times \{FALSE\}$

int1_10 : $envData := (ND \setminus \{Sink\}) \times \{0\}$

int1_11 : $ctlSensedFlg := (ND \setminus \{Sink\}) \times \{FALSE\}$

int1_12 : $floodFlg := ND \times \{FALSE\}$

inv1_20 : $recLostPkts := \varnothing$

int2_1 : $sentUp := \varnothing$

int2_2 : $sentDown := \varnothing$

int2_3 : $channel := \varnothing$

int2_10 : $envNeighbours := \varnothing$

int2_11 : $wsnLinks := \varnothing$

int2_12 : $crashedLinks := \varnothing$

int2_13 : $dataSeqNo := ND \times \{0\}$

int2_14 : $floodSeqNo := ND \times \{0\}$

int2_15 : $nbHops := \varnothing$

int2_16 : $pktSeqNo := \varnothing$

int2_17 : $pktSrc := \varnothing$

int2_18 : $pktFwdr := \varnothing$

int2_19 : $pktData := \varnothing$

int2_20 : $pktNbHops := \varnothing$

int2_21 : $vPktSeqNo := \varnothing$

int2_22 : $vPktSrc := \varnothing$

int2_23 : $vPktFwdr := \varnothing$

int2_24 : $vPktData := \varnothing$

int2_25 : $vPktNbHops := \varnothing$

end

Event *add_link* $\widehat{=}$

//(env) channel sets a topology

any

x

y

where

g1 : $x \in ND \wedge y \in ND$

g2 : $x \mapsto y \notin wsnLinks$

g3 : $x \neq y$

then

a1 : $wsnLinks := wsnLinks \cup \{x \mapsto y\}$

end

Event *disconnect_link* $\widehat{=}$

//(env) disconnecting a link

any

x

y

where

g1 : $x \in ND \wedge y \in ND$

g2 : $x \mapsto y \in wsnLinks$

g3 : $x \mapsto y \notin crashedLinks$

then

a1 : $crashedLinks := crashedLinks \cup \{x \mapsto y\}$

a2 : $wsnLinks := wsnLinks \setminus \{x \mapsto y\}$

end

Event *recover_link* $\widehat{=}$

//(env) recovering a link

any

x

y

where

g1 : $x \in ND \wedge y \in ND$

g2 : $x \mapsto y \in crashedLinks$

g3 : $x \mapsto y \notin wsnLinks$

then

a1 : $wsnLinks := wsnLinks \cup \{x \mapsto y\}$

a2 : $crashedLinks := crashedLinks \setminus \{x \mapsto y\}$

end

Event *start_sensing* $\widehat{=}$

//(env) activating each node to sense data

any

x

sd

where

g1 : $x \in ND \setminus \{Sink\}$

g2 : $envSensedFlg(x) = FALSE$

g3 : $sd \in randomFn$

then

a1 : $envSensedFlg(x) := TRUE$

a2 : $envData(x) := sd$

end

Event *sensing* $\widehat{=}$

//(shd) each node senses a data from environment

any

x

sf

sd

where

g1 : $x \in ND \setminus \{Sink\}$

g2 : $sf = envSensedFlg(x)$

g3 : $envSensedFlg(x) = TRUE$

g4 : $ctlSensedFlg(x) = FALSE$

g5 : $sd = envData(x)$

then

a1 : $ctlSensedFlg(x) := sf$

a2 : $senseBuff(x) := senseBuff(x) \cup \{sd\}$

end

Event *finish_sensing* $\widehat{=}$

//(shd) finishing sensing data process

refines *clear_sensingNDs*

any

x

where

g1 : $x \in dom(envSensedFlg)$

g2 : $envSensedFlg(x) = TRUE$

g3 : $ndBuff = \varnothing \wedge (sentDown \cup sentUp) = \varnothing$

g4 : $ctlSensedFlg(x) = TRUE$

g5 : $sensingNDs = \{x\}$

then

a1 : $ctlSensedFlg(x) := FALSE$

a2 : $envSensedFlg(x) := FALSE$
a3 : $sensingNDs := \varnothing$

end

Event *create_dataPkt* $\widehat{=}$

//(ctl) creating a data packet

refines *create_dataPkt*

any

s
pkt
sd
sno
nbh

where

g1 : $pkt \in PKT \wedge s \in ND \setminus \{Sink\}$
g2 : $pkt \notin ran(ndBuff \cup (sentUp \cup sentDown))$
g3 : $s \notin sensingNDs \wedge pkt \notin sensedPkts \cup floodedPkts$ //has not been created yet
g4 : $s = initialSrcAddr(pkt)$
g5 : $s \in dom(floodTbl) \wedge pkt \notin floodTbl(s)$ //is not a duplicated packet
g6 : $type(pkt) = DATA$
g7 : $ctlSensedFlg(s) = TRUE$
g8 : $senseBuff \neq \varnothing \wedge sd \in senseBuff(s)$
g9 : $sno = dataSeqNo(s) + 1 \wedge s \in dom(dataSeqNo)$ //identifing sequence number
g10 : $nbh \in \mathbb{Z} \wedge nbh = -1$ //starting with hop count to be -1
g11 : $s \mapsto \{pkt \mapsto nbh\} \notin nbHops$
g12 : $pkt \notin dom(pktSeqNo)$
g13 : $pkt \notin dom(pktSrc)$
g14 : $pkt \notin dom(pktFwdr)$
g15 : $pkt \notin dom(pktData)$
g16 : $pkt \notin dom(pktNbHops)$

then

a1 : $sensingNDs := sensingNDs \cup \{s\}$
a2 : $sensedPkts := sensedPkts \cup \{pkt\}$
a3 : $ndBuff := ndBuff \cup \{s \mapsto pkt\}$
a4 : $floodTbl(s) := floodTbl(s) \cup \{pkt\}$
a5 : $dataSeqNo := dataSeqNo \Leftarrow \{s \mapsto sno\}$
a6 : $nbHops := nbHops \cup \{s \mapsto \{pkt \mapsto nbh\}\}$
a7 : $senseBuff(s) := senseBuff(s) \setminus \{sd\}$
a8 : $pktSeqNo := pktSeqNo \Leftarrow \{pkt \mapsto sno\}$
a9 : $pktSrc := pktSrc \cup \{pkt \mapsto s\}$
a10 : $pktFwdr := pktFwdr \cup \{pkt \mapsto s\}$
a11 : $pktData := pktData \cup \{pkt \mapsto sd\}$

a12 : $pktNbHops := pktNbHops \cup \{pkt \mapsto nbh\}$

end

Event *create_controlPkt* $\widehat{=}$

//(ctl) creating a control packet

refines *create_controlPkt*

any

s

pkt

sno

nbh

where

g1 : $pkt \in PKT$

g2 : $pkt \notin ran(ndBuff \cup (sentUp \cup sentDown))$

g3 : $pkt \notin (sensedPkts \cup floodedPkts)$ //has not been created yet

g4 : $s = initialSrcAddr(pkt)$

g5 : $s \in dom(floodTbl) \wedge pkt \notin floodTbl(s)$

g6 : $type(pkt) \in CONTROL$

g7 : $floodFlg(s) = TRUE$

g8 : $sno = floodSeqNo(s) + 1 \wedge s \in dom(floodSeqNo)$ //identifing sequence number

g9 : $nbh \in \mathbb{Z} \wedge nbh = -1$ //starting with hop count to be -1

g10 : $s \mapsto \{pkt \mapsto nbh\} \notin nbHops$

g11 : $pkt \notin dom(pktSeqNo)$

g12 : $pkt \notin dom(pktSrc)$

g13 : $pkt \notin dom(pktFwdr)$

g14 : $pkt \notin dom(pktData)$

g15 : $pkt \notin dom(pktNbHops)$

then

a1 : $floodedPkts := floodedPkts \cup \{pkt\}$

a2 : $ndBuff := ndBuff \cup \{s \mapsto pkt\}$

a3 : $floodTbl(s) := floodTbl(s) \cup \{pkt\}$

a4 : $floodSeqNo := floodSeqNo \Leftarrow \{s \mapsto sno\}$

a5 : $nbHops := nbHops \cup \{s \mapsto \{pkt \mapsto nbh\}\}$

a6 : $pktSeqNo := pktSeqNo \Leftarrow \{pkt \mapsto sno\}$

a7 : $pktFwdr := pktFwdr \cup \{pkt \mapsto s\}$

a8 : $pktSrc := pktSrc \cup \{pkt \mapsto s\}$

a9 : $pktData := pktData \cup \{pkt \mapsto 0\}$

a10 : $pktNbHops := pktNbHops \cup \{pkt \mapsto nbh\}$

end

Event *start_tx* $\widehat{=}$

//(ctl) starting transmiting a packet to a channel

refines *start_tx*

any

x

pkt

nbh

where

g1 : $x \mapsto pkt \in ndBuff$ //packet in waiting buffer

g2 : $x \mapsto pkt \notin sentDown$ //but has not been sent down to a channel

g3 : $x \mapsto pkt \notin sentUp$ //and has not been sent up from a channel

g4 : $pkt \in dom(pktFwdr)$

g5 : $x \mapsto \{pkt \mapsto nbh\} \in nbHops$

g6 : $pkt \in dom(pktNbHops)$

then

a1 : $ndBuff := ndBuff \setminus \{x \mapsto pkt\}$

a2 : $sentDown := sentDown \cup \{x \mapsto pkt\}$

a3 : $pktFwdr := pktFwdr \lhd\!\!\!- \{pkt \mapsto x\}$

a4 : $pktNbHops := pktNbHops \lhd\!\!\!- \{pkt \mapsto nbh\}$

a5 : $nbHops := nbHops \setminus \{x \mapsto \{pkt \mapsto nbh\}\}$

end

Event *send_down* $\widehat{=}$

//(shd) sending a packet down to a channel

any

cn

pkt

sno

src

$data$

$fwdr$

nbh

where

g1 : $cn \mapsto pkt \in sentDown$

g2 : $cn \mapsto pkt \notin channel$

g3 : $pkt \in dom(pktSeqNo) \wedge sno = pktSeqNo(pkt)$

g4 : $pkt \in dom(pktSrc) \wedge src = pktSrc(pkt)$

g5 : $pkt \in dom(pktFwdr) \wedge fwdr = pktFwdr(pkt)$

g6 : $pkt \in dom(pktData) \wedge data = pktData(pkt)$

g7 : $pkt \in dom(pktNbHops) \wedge nbh = pktNbHops(pkt)$

g8 : $pkt \notin dom(vPktSeqNo)$

g9 : $pkt \notin dom(vPktFwdr)$

g10 : $pkt \notin dom(vPktSrc)$

g11 : $pkt \notin dom(vPktData)$

g12 : $pkt \notin dom(vPktNbHops)$

then

a1 : $channel := channel \cup \{cn \mapsto pkt\}$

a2 : $vPktSeqNo := vPktSeqNo \cup \{pkt \mapsto sno\}$

a3 : $vPktSrc := vPktSrc \cup \{pkt \mapsto src\}$

a4 : $vPktFwdr := vPktFwdr \cup \{pkt \mapsto fwdr\}$

a5 : $vPktData := vPktData \cup \{pkt \mapsto data\}$

a6 : $vPktNbHops := vPktNbHops \cup \{pkt \mapsto nbh\}$

a7 : $pktSeqNo := \{pkt\} \lhd pktSeqNo$

a8 : $pktSrc := \{pkt\} \lhd pktSrc$

a9 : $pktFwdr := \{pkt\} \lhd pktFwdr$

a10 : $pktData := \{pkt\} \lhd pktData$

a11 : $pktNbHops := \{pkt\} \lhd pktNbHops$

end

Event *find_neighbours* $\widehat{=}$

//(env) channel finds the neighbour list of the forwarder

any

s

nbs

f

pkt

where

g1 : $channel \neq \varnothing$

g2 : $pkt \in ran(channel)$

g3 : $s = initialSrcAddr(pkt)$

g4 : $f \mapsto pkt \in channel$

g5 : $nbs \subseteq ND \wedge nbs = wsnLinks[\{f\}] \wedge nbs \neq \varnothing$

//identifying all neighbours of current forwarder from network topology

g6 : $envNeighbours = \varnothing$

then

a1 : $envNeighbours := envNeighbours \cup (\{pkt\} \times nbs)$

end

Event *assign_forwarder* $\widehat{=}$

//(env) channel assigns specific forwarder - preparing for unicast for later refinement

any

s

nb

pkt

f

nbs

where

g1 : $channel \neq \varnothing$

g2 : $pkt \in ran(channel)$

g3 : $s = initialSrcAddr(pkt)$

g4 : $f \mapsto pkt \in channel$

g5 : $nbs = wsnLinks[\{f\}] \wedge nbs \neq \varnothing$

g6 : $nb \in nbs$

//creating a choice of specific neighbours of current forwarder from network topology

then

a1 : $envNeighbours := envNeighbours \cup \{pkt \mapsto nb\}$

end

Event *send_up* $\widehat{=}$

//(shd) channel sends the neighbour information

refines *indicate_neighbours*

any

pkt

$nbrs$

f

sno

src

$fwdr$

$data$

nbh

where

g1 : $pkt \in ran(sentDown)$

g2 : $f \mapsto pkt \in sentDown$

g3 : $nbrs \in \{n | n \in (\mathbb{P}(ND)) \cup \{\{FAILED\_XMIT\}\}\}$

g4 : $nbrs \neq \varnothing$

g5 : $ran(\{pkt\} \lhd envNeighbours) \in (\mathbb{P}(ND) \cup \{\{FAILED\_XMIT\}\})$

g6 : $nbrs = ran(\{pkt\} \lhd envNeighbours) \wedge nbrs \neq \varnothing$

//retrieving neighbours from environment

g7 : $pkt \notin dom(ctlNeighbours)$

g8 : $f \mapsto pkt \notin sentUp$

g9 : $f \mapsto pkt \in channel$

g10 : $pkt \in dom(vPktSeqNo) \wedge sno = vPktSeqNo(pkt)$

g11 : $pkt \in dom(vPktSrc) \wedge src = vPktSrc(pkt)$

g12 : $pkt \in dom(vPktFwdr) \wedge fwdr = vPktFwdr(pkt)$

g13 : $pkt \in dom(vPktData) \wedge data = vPktData(pkt)$

g14 : $pkt \in dom(vPktNbHops) \wedge nbh = vPktNbHops(pkt)$

g15 : $pkt \notin dom(pktSeqNo)$

g16 : $pkt \notin dom(pktSrc)$
g17 : $pkt \notin dom(pktFwdr)$
g18 : $pkt \notin dom(pktData)$
g19 : $pkt \notin dom(pktNbHops)$

then

a1 : $ctlNeighbours := ctlNeighbours \cup (\{pkt\} \times nbrs)$
//updating neighbours from environment to controller
a2 : $envNeighbours := \varnothing$
a3 : $channel := channel \setminus \{f \mapsto pkt\}$
a4 : $sentDown := sentDown \setminus \{f \mapsto pkt\}$
a5 : $sentUp := sentUp \cup \{f \mapsto pkt\}$
a6 : $pktSeqNo := pktSeqNo \cup \{pkt \mapsto sno\}$
a7 : $pktSrc := pktSrc \cup \{pkt \mapsto src\}$
a8 : $pktFwdr := pktFwdr \cup \{pkt \mapsto fwdr\}$
a9 : $pktData := pktData \cup \{pkt \mapsto data\}$
a10 : $pktNbHops := pktNbHops \cup \{pkt \mapsto nbh\}$
a11 : $vPktSeqNo := \{pkt\} \lhd vPktSeqNo$
a12 : $vPktSrc := \{pkt\} \lhd vPktSrc$
a13 : $vPktFwdr := \{pkt\} \lhd vPktFwdr$
a14 : $vPktData := \{pkt\} \lhd vPktData$
a15 : $vPktNbHops := \{pkt\} \lhd vPktNbHops$

end

Event *receive_pkt* $\widehat{=}$
//(ctl) each neighbour node recieves a transmitted packet
refines *receive_pkt*

any

s
nb
pkt
nbh

where

g1 : $pkt \notin ran(sentDown) \wedge pkt \in ran(sentUp)$
g2 : $pkt \mapsto nb \in ctlNeighbours$
g3 : $nb \mapsto pkt \notin ndBuff$
g4 : $nb \mapsto pkt \notin (sentUp \cup sentDown)$
g5 : $s = initialSrcAddr(pkt) \wedge s \neq nb$
g6 : $nb \neq Sink$
g7 : $nb \in dom(floodTbl) \wedge pkt \notin floodTbl(nb)$ //is not a duplicated packet
g7 : $pkt \in dom(pktNbHops) \wedge nbh = pktNbHops(pkt) + 1$

then

a1 : $ndBuff := ndBuff \cup \{nb \mapsto pkt\}$

a2 : $ctlNeighbours := ctlNeighbours \setminus \{pkt \mapsto nb\}$
a3 : $floodTbl(nb) := floodTbl(nb) \cup \{pkt\}$
a4 : $nbHops := nbHops \cup \{nb \mapsto \{pkt \mapsto nbh\}\}$

end

Event *receive_dup_pkt* $\widehat{=}$

//(ctl) each neighbour node discards a duplicated packet

extends *receive_dup_pkt*

any

nb
pkt

where

g1 : $pkt \in dom(ctlNeighbours)$
g2 : $pkt \mapsto nb \in ctlNeighbours$
g3 : $nb \in dom(floodTbl) \wedge pkt \in floodTbl(nb)$ //is a duplicated packet

then

a1 : $ctlNeighbours := ctlNeighbours \setminus \{pkt \mapsto nb\}$

end

Event *sink_recv_pkt* $\widehat{=}$

//(ctl) a sink receives a packet

refines *sink_recv_pkt*

any

s
pkt
nb

where

g1 : $s = initialSrcAddr(pkt)$
g2 : $s \neq Sink$
g3 : $pkt \in ran(sentDown \cup sentUp) \setminus sinkBuff$
g4 : $nb = Sink$
g5 : $pkt \mapsto nb \in ctlNeighbours$
g6 : $nb \in dom(floodTbl) \wedge pkt \notin floodTbl(nb)$

then

a1 : $sinkBuff := sinkBuff \cup \{pkt\}$
a2 : $ctlNeighbours := ctlNeighbours \setminus \{pkt \mapsto nb\}$
a3 : $floodTbl(nb) := floodTbl(nb) \cup \{pkt\}$

end

Event *finish_tx_pkt* $\widehat{=}$

//(ctl) finishing packet transmission

refines *finish_tx_pkt*

any

f

pkt

where

g1 : $\{pkt\} \lhd ctlNeighbours = \varnothing$

g2 : $pkt \in ran(sentUp) \wedge pkt \notin ran(sentDown) \wedge pkt \in ran(ndBuff)$

g3 : $f \in dom(sentUp) \wedge f \mapsto pkt \in sentUp$

then

a1 : $sentUp := sentUp \setminus \{f \mapsto pkt\}$

end

Event *final_tx_pkt* $\widehat{=}$

//(ctl) the transmitted packet is removed from middleware indicating completion of transmission mechanism

refines *final_tx_pkt*

any

pkt

where

g1 : $\{pkt\} \lhd ctlNeighbours = \varnothing$

g2 : $pkt \in ran(sentUp) \wedge pkt \notin ran(sentDown) \wedge pkt \notin ran(ndBuff)$

then

a1 : $sentUp := sentUp \rhd \{pkt\}$

end

Event *lose_all_neighbours* $\widehat{=}$

//(env) channel loses all neighbours

any

s

nbs

f

pkt

where

g1 : $channel \neq \varnothing$

g2 : $pkt \in ran(channel)$

g3 : $s = initialSrcAddr(pkt)$

g4 : $f \mapsto pkt \in channel$

g5 : $nbs = wsnLinks[\{f\}] \wedge nbs = \varnothing$ //there are no neighbours of a forwarder

then

a1 : $envNeighbours := envNeighbours \cup \{pkt \mapsto FAILED\_XMIT\}$

end

Event *lose_pkt* $\widehat{=}$

//(ctl) each node records a lost packet

refines *lose_pkt*

any

f

pkt

where

g1 : $pkt \in ran(sentUp) \wedge pkt \notin ran(sentDown) \wedge pkt \notin ran(ndBuff)$

g2 : $pkt \notin ran(recLostPkts)$

g3 : $f \mapsto pkt \in sentUp \wedge f \mapsto pkt \notin sentDown$

g4 : $f \mapsto pkt \notin recLostPkts$

g5 : $ran(\{pkt\} \lhd ctlNeighbours) = \{FAILED\_XMIT\}$

then

a1 : $sentUp := sentUp \unrhd \{pkt\}$

a2 : $recLostPkts := recLostPkts \cup \{f \mapsto pkt\}$

end

END

A.4 Third refinement: neighbour table management and link quality estimation

A.4.1 Context:C3

CONTEXT C3

EXTENDS C2

CONSTANTS

EST_RATIO

$LIVELINESS$

$ROUTE$

$BEACON$

AXIOMS

axm3_1 : $EST\_RATIO \in \mathbb{N}_1$ //estimation ratio

axm3_2 : $LIVELINESS \in \mathbb{N}_1$ //liveliness threshold

axm3_3 : $partition(CONTROL, \{ROUTE\}, \{BEACON\})$

//two control packet types: route and beacon packets

END

A.4.2 Machine:M3

MACHINE M3

REFINES M2

SEES C3

VARIABLES

$sinkBuff$ //(ctl) a sink buffer

$ndBuff$ //(ctl) nodes' waiting buffer

$sensingNDs$ //(ctl) sensing nodes

$sensedPkts$ //(ctl) sensed packets
$senseBuff$ //(ctl) sense buffer
$floodedPkts$ //(ctl) flooded packets
$floodTbl$ //(ctl)historical received packets used for detect duplicated packets
$ctlNeighbours$ //(ctl) neighbour lists
$recLostPkts$ //(ctl) recorded lost packet at each node
$envSensedFlg$ //(env) sensing period
$envData$ //(env) environmental/conditional data
$ctlSensedFlg$ //(ctl) sensing period
$sentDown$ //(ctl) transmitted packets down to channel
$sentUp$ //(ctl) transmitted packets up from channel
$channel$ //(env) channel
$envNeighbours$ //(env) neighbour list
$wsnLinks$ //(env) dynamic WSN links
$floodFlg$ //(ctl) flooded flag
$dataSeqNo$ //(ctl) local nodes' current data sequence number at application layer
$floodSeqNo$ //(ctl) local nodes' current broadcast sequence number at application layer
$nbHops$ //(ctl) neighbour's hop count
$crashedLinks$ //(env) crashed WSN Links
$pktSrc$ //(ctl) src address attached in transmitted packet
$pktSeqNo$ //(ctl) sequence number attached in transmitted packet
$pktFwdr$ //(ctl) fowarder attached in transmitted packet
$pktData$ //(ctl) data attached in transmitted packet
$pktNbHops$ //(ctl) hop count attached in transmitted packet
$vPktSeqNo$ //(env) virtual packet at channel
$vPktSrc$ //(env) virtual packet at channel
$vPktFwdr$ //(env) virtual packet at channel
$vPktData$ //(env) virtual packet at channel
$vPktNbHops$ //(env) virtual packet at channel
$totalSentBcon$ //(ctl) number of sent beacon packets, used for cost-estimation
$netSeqNo$ //(ctl) link sequence no in a packet, used for cost-estimation
$linkSeqNo$ //(ctl) link sequence number of each node
$updateNbrs$ //(ctl) for updating neighbours in the neighbour table
$neighbourTbl$ //(ctl)neighbour table structure
$missed$ //(ctl) neighbour table: field missed (number of lost packets)
$received$ //(ctl) neighbour table: field received (number of received packets)
$lastSeqno$ //(ctl) neighbour table: field lastSeqno in neighbour table
$receiveEst$ //(ctl) neighbour talble: field receiveEst (reception link quality)
$estNDs$ //(ctl) estimated nodes
$estNbrs$ //(ctl) estimated neighbours

INVARIANTS

inv3_1 : $neighbourTbl \in ND \leftrightarrow ND$
inv3_2 : $missed \in neighbourTbl \rightarrow \mathbb{N}$
inv3_3 : $received \in neighbourTbl \rightarrow \mathbb{N}$
inv3_4 : $lastSeqno \in neighbourTbl \rightarrow \mathbb{N}$
inv3_5 : $receiveEst \in neighbourTbl \rightarrow \mathbb{N}$
inv3_6 : $totalSentBcon \in ND \rightarrow \mathbb{N}$
inv3_7 : $netSeqNo \in PKT \rightarrow \mathbb{N}$
inv3_8 : $updateNbrs \in ND \leftrightarrow ND$
inv3_9 : $linkSeqNo \in ND \rightarrow \mathbb{N}$
int3_10 : $estNDs \subseteq ND$
inv3_11 : $estNbrs \subseteq neighbourTbl$

EVENTS

Initialisation

extended

begin

int0_3 : $sinkBuff := \varnothing$
int1_1 : $ndBuff := \varnothing$
int1_3 : $sensingNDs := \varnothing$
int1_4 : $sensedPkts := \varnothing$
int1_5 : $senseBuff := (ND \setminus \{Sink\}) \times \{\varnothing\}$
int1_6 : $floodedPkts := \varnothing$
int1_7 : $floodTbl := ND \times \{\varnothing\}$
int1_8 : $ctlNeighbours := \varnothing$
int1_9 : $envSensedFlg := (ND \setminus \{Sink\}) \times \{FALSE\}$
int1_10 : $envData := (ND \setminus \{Sink\}) \times \{0\}$
int1_11 : $ctlSensedFlg := (ND \setminus \{Sink\}) \times \{FALSE\}$
int1_12 : $floodFlg := ND \times \{FALSE\}$
inv1_20 : $recLostPkts := \varnothing$
int2_1 : $sentUp := \varnothing$
int2_2 : $sentDown := \varnothing$
int2_3 : $channel := \varnothing$
int2_10 : $envNeighbours := \varnothing$
int2_11 : $wsnLinks := \varnothing$
int2_12 : $crashedLinks := \varnothing$
int2_13 : $dataSeqNo := ND \times \{0\}$
int2_14 : $floodSeqNo := ND \times \{0\}$
int2_15 : $nbHops := \varnothing$
int2_16 : $pktSeqNo := \varnothing$
int2_17 : $pktSrc := \varnothing$
int2_18 : $pktFwdr := \varnothing$
int2_19 : $pktData := \varnothing$

int2_20 : $pktNbHops := \varnothing$
int2_21 : $vPktSeqNo := \varnothing$
int2_22 : $vPktSrc := \varnothing$
int2_23 : $vPktFwdr := \varnothing$
int2_24 : $vPktData := \varnothing$
int2_25 : $vPktNbHops := \varnothing$
int3_1 : $totalSentBcon := ND \times \{0\}$
int3_2 : $netSeqNo := PKT \times \{0\}$
int3_3 : $updateNbrs := \varnothing$
int3_4 : $neighbourTbl := \varnothing$
int3_5 : $missed := \varnothing$
int3_6 : $received := \varnothing$
int3_7 : $lastSeqno := \varnothing$
int3_8 : $receiveEst := \varnothing$
int3_9 : $linkSeqNo := ND \times \{0\}$
int3_11 : $estNDs := \varnothing$
int3_12 : $estNbrs := \varnothing$

end

Event *add_link* $\widehat{=}$

//(env) channel sets a topology

extends *add_link*

any

x

y

where

g1 : $x \in ND \wedge y \in ND$
g2 : $x \mapsto y \notin wsnLinks$
g3 : $x \neq y$

then

a1 : $wsnLinks := wsnLinks \cup \{x \mapsto y\}$

end

Event *disconnect_link* $\widehat{=}$

//(env) disconnecting a link

extends *disconnect_link*

any

x

y

where

g1 : $x \in ND \wedge y \in ND$
g2 : $x \mapsto y \in wsnLinks$
g3 : $x \mapsto y \notin crashedLinks$

then

a1 : $crashedLinks := crashedLinks \cup \{x \mapsto y\}$

a2 : $wsnLinks := wsnLinks \setminus \{x \mapsto y\}$

end

Event *recover_link* $\widehat{=}$

//(env) recovering a link

extends *recover_link*

any

x

y

where

g1 : $x \in ND \wedge y \in ND$

g2 : $x \mapsto y \in crashedLinks$

g3 : $x \mapsto y \notin wsnLinks$

g4 : $x \neq y$

then

a1 : $wsnLinks := wsnLinks \cup \{x \mapsto y\}$

a2 : $crashedLinks := crashedLinks \setminus \{x \mapsto y\}$

end

Event *start_sensing* $\widehat{=}$

//(env) activating each node to sense data

extends *start_sensing*

any

x

sd

where

g1 : $x \in ND \setminus \{Sink\}$

g2 : $envSensedFlg(x) = FALSE$

g3 : $sd \in randomFn$

then

a1 : $envSensedFlg(x) := TRUE$

a2 : $envData(x) := sd$

end

Event *sensing* $\widehat{=}$

//(shd) each node senses a data from environment

extends *sensing*

any

x

sf

sd

where

g1 : $x \in ND \setminus \{Sink\}$
g2 : $sf = envSensedFlg(x)$
g3 : $envSensedFlg(x) = TRUE$
g4 : $ctlSensedFlg(x) = FALSE$
g5 : $sd = envData(x)$

then

a1 : $ctlSensedFlg(x) := sf$
a2 : $senseBuff(x) := senseBuff(x) \cup \{sd\}$

end

Event *finish_sensing* $\widehat{=}$

//(shd) finishing sensing data process

extends *finish_sensing*

any

x

where

g1 : $x \in dom(envSensedFlg)$
g2 : $envSensedFlg(x) = TRUE$
g3 : $ndBuff = \varnothing \wedge (sentDown \cup sentUp) = \varnothing$
g4 : $ctlSensedFlg(x) = TRUE$
g5 : $sensingNDs = \{x\}$

then

a1 : $ctlSensedFlg(x) := FALSE$
a2 : $envSensedFlg(x) := FALSE$
a3 : $sensingNDs := \varnothing$

end

Event *create_dataPkt* $\widehat{=}$

//(ctl) creating a data packet

extends *create_dataPkt*

any

s
pkt
sd
sno
nbh

where

g1 : $pkt \in PKT \wedge s \in ND \setminus \{Sink\}$
g2 : $pkt \notin ran(ndBuff \cup (sentUp \cup sentDown))$
g3 : $s \notin sensingNDs \wedge pkt \notin sensedPkts \cup floodedPkts$
g4 : $s = initialSrcAddr(pkt)$
g5 : $s \in dom(floodTbl) \wedge pkt \notin floodTbl(s)$
g6 : $type(pkt) = DATA$

g7 : $ctlSensedFlg(s) = TRUE$
g8 : $senseBuff \neq \varnothing \wedge sd \in senseBuff(s)$
g9 : $sno = dataSeqNo(s) + 1 \wedge s \in dom(dataSeqNo)$
g10 : $nbh \in \mathbb{Z} \wedge nbh = -1$
g11 : $s \mapsto \{pkt \mapsto nbh\} \notin nbHops$
g12 : $pkt \notin dom(pktSeqNo)$
g13 : $pkt \notin dom(pktSrc)$
g14 : $pkt \notin dom(pktFwdr)$
g15 : $pkt \notin dom(pktData)$
g16 : $pkt \notin dom(pktNbHops)$

then

a1 : $sensingNDs := sensingNDs \cup \{s\}$
a2 : $sensedPkts := sensedPkts \cup \{pkt\}$
a3 : $ndBuff := ndBuff \cup \{s \mapsto pkt\}$
a4 : $floodTbl(s) := floodTbl(s) \cup \{pkt\}$
a5 : $dataSeqNo := dataSeqNo \Leftarrow \{s \mapsto sno\}$
a6 : $nbHops := nbHops \cup \{s \mapsto \{pkt \mapsto nbh\}\}$
a7 : $senseBuff(s) := senseBuff(s) \setminus \{sd\}$
a8 : $pktSeqNo := pktSeqNo \Leftarrow \{pkt \mapsto sno\}$
a9 : $pktSrc := pktSrc \cup \{pkt \mapsto s\}$
a10 : $pktFwdr := pktFwdr \cup \{pkt \mapsto s\}$
a11 : $pktData := pktData \cup \{pkt \mapsto sd\}$
a12 : $pktNbHops := pktNbHops \cup \{pkt \mapsto nbh\}$

end

Event *create_controlPkt* $\widehat{=}$

//(ctl) creating a control packet

extends *create_controlPkt*

any

s
pkt
sno
nbh

where

g1 : $pkt \in PKT$
g2 : $pkt \notin ran(ndBuff \cup (sentUp \cup sentDown))$
g3 : $pkt \notin (sensedPkts \cup floodedPkts)$
g4 : $s = initialSrcAddr(pkt)$
g5 : $s \in dom(floodTbl) \wedge pkt \notin floodTbl(s)$
g6 : $type(pkt) \in CONTROL$
g7 : $floodFlg(s) = TRUE$
g8 : $sno = floodSeqNo(s) + 1 \wedge s \in dom(floodSeqNo)$

g9 : $nbh \in \mathbb{Z} \wedge nbh = -1$
g10 : $s \mapsto \{pkt \mapsto nbh\} \notin nbHops$
g12 : $pkt \notin dom(pktSeqNo)$
g13 : $pkt \notin dom(pktSrc)$
g14 : $pkt \notin dom(pktFwdr)$
g15 : $pkt \notin dom(pktData)$
g16 : $pkt \notin dom(pktNbHops)$

then

a1 : $floodedPkts := floodedPkts \cup \{pkt\}$
a2 : $ndBuff := ndBuff \cup \{s \mapsto pkt\}$
a3 : $floodTbl(s) := floodTbl(s) \cup \{pkt\}$
a4 : $floodSeqNo := floodSeqNo \lhd\!\!\!- \{s \mapsto sno\}$
a5 : $nbHops := nbHops \cup \{s \mapsto \{pkt \mapsto nbh\}\}$
a6 : $pktSeqNo := pktSeqNo \lhd\!\!\!- \{pkt \mapsto sno\}$
a7 : $pktFwdr := pktFwdr \cup \{pkt \mapsto s\}$
a8 : $pktSrc := pktSrc \cup \{pkt \mapsto s\}$
a9 : $pktData := pktData \cup \{pkt \mapsto 0\}$
a10 : $pktNbHops := pktNbHops \cup \{pkt \mapsto nbh\}$

end

Event *start_tx_dataPkt* $\widehat{=}$

//(ctl) starting transmiting a data packet to channel

extends *start_tx*

any

x

pkt

nbh

where

g1 : $x \mapsto pkt \in ndBuff$
g2 : $x \mapsto pkt \notin sentDown$
g3 : $x \mapsto pkt \notin sentUp$
g5 : $pkt \in dom(pktFwdr)$
g6 : $x \mapsto \{pkt \mapsto nbh\} \in nbHops$
g7 : $pkt \in dom(pktNbHops)$
g8 : $type(pkt) = DATA$

then

a1 : $ndBuff := ndBuff \setminus \{x \mapsto pkt\}$
a2 : $sentDown := sentDown \cup \{x \mapsto pkt\}$
a3 : $pktFwdr := pktFwdr \lhd\!\!\!- \{pkt \mapsto x\}$
a4 : $pktNbHops := pktNbHops \lhd\!\!\!- \{pkt \mapsto nbh\}$
a5 : $nbHops := nbHops \setminus \{x \mapsto \{pkt \mapsto nbh\}\}$

end

Event *start_tx_bconPkt* $\widehat{=}$

//(ctl) starting transmiting a beacon packet to a channel

extends *start_tx*

any

x

pkt

nbh

$lsno$

where

g1 : $x \mapsto pkt \in ndBuff$

g2 : $x \mapsto pkt \notin sentDown$

g3 : $x \mapsto pkt \notin sentUp$

g4 : $pkt \in dom(pktFwdr)$

g5 : $x \mapsto \{pkt \mapsto nbh\} \in nbHops$

g6 : $pkt \in dom(pktNbHops)$

g7 : $type(pkt) = BEACON$

g8 : $x \in dom(totalSentBcon) \wedge totalSentBcon(x) \geq 0$

g9 : $lsno = linkSeqNo(x) + 1 \wedge x \in dom(linkSeqNo)$

then

a1 : $ndBuff := ndBuff \setminus \{x \mapsto pkt\}$

a2 : $sentDown := sentDown \cup \{x \mapsto pkt\}$

a3 : $pktFwdr := pktFwdr \lhd\!\!\!+ \{pkt \mapsto x\}$

a4 : $pktNbHops := pktNbHops \lhd\!\!\!+ \{pkt \mapsto nbh\}$

a5 : $nbHops := nbHops \setminus \{x \mapsto \{pkt \mapsto nbh\}\}$

a6 : $totalSentBcon(x) := totalSentBcon(x) + 1$

a7 : $linkSeqNo := linkSeqNo \lhd\!\!\!+ \{x \mapsto lsno\}$

a8 : $netSeqNo(pkt) := lsno$

//update the current sequence number into a control packet

end

Event *start_tx_controlPkt* $\widehat{=}$

//(ctl) starting transmiting a control packet to a channel

extends *start_tx*

any

x

pkt

nbh

$lsno$

where

g1 : $x \mapsto pkt \in ndBuff$

g2 : $x \mapsto pkt \notin sentDown$

g3 : $x \mapsto pkt \notin sentUp$

g4 : $pkt \in dom(pktFwdr)$
g5 : $x \mapsto \{pkt \mapsto nbh\} \in nbHops$
g6 : $pkt \in dom(pktNbHops)$
g7 : $type(pkt) \neq BEACON$
g8 : $lsno \in \mathbb{N}$

then

a1 : $ndBuff := ndBuff \setminus \{x \mapsto pkt\}$
a2 : $sentDown := sentDown \cup \{x \mapsto pkt\}$
a3 : $pktFwdr := pktFwdr \lhd\!\!\!- \{pkt \mapsto x\}$
a4 : $pktNbHops := pktNbHops \lhd\!\!\!- \{pkt \mapsto nbh\}$
a5 : $nbHops := nbHops \setminus \{x \mapsto \{pkt \mapsto nbh\}\}$
a6 : $netSeqNo(pkt) := lsno$

end

Event *send_down* $\widehat{=}$

//(shd) sending a packet down to a channel

extends *send_down*

any

cn
pkt
sno
src
$data$
$fwdr$
nbh

where

g1 : $cn \mapsto pkt \in sentDown$
g2 : $cn \mapsto pkt \notin channel$
g3 : $pkt \in dom(pktSeqNo) \wedge sno = pktSeqNo(pkt)$
g4 : $pkt \in dom(pktSrc) \wedge src = pktSrc(pkt)$
g5 : $pkt \in dom(pktFwdr) \wedge fwdr = pktFwdr(pkt)$
g6 : $pkt \in dom(pktData) \wedge data = pktData(pkt)$
g7 : $pkt \in dom(pktNbHops) \wedge nbh = pktNbHops(pkt)$
g8 : $pkt \notin dom(vPktSeqNo)$
g9 : $pkt \notin dom(vPktFwdr)$
g10 : $pkt \notin dom(vPktSrc)$
g11 : $pkt \notin dom(vPktData)$
g12 : $pkt \notin dom(vPktNbHops)$

then

a1 : $channel := channel \cup \{cn \mapsto pkt\}$
a2 : $vPktSeqNo := vPktSeqNo \cup \{pkt \mapsto sno\}$
a3 : $vPktSrc := vPktSrc \cup \{pkt \mapsto src\}$

a4 : $vPktFwdr := vPktFwdr \cup \{pkt \mapsto fwdr\}$
a5 : $vPktData := vPktData \cup \{pkt \mapsto data\}$
a6 : $vPktNbHops := vPktNbHops \cup \{pkt \mapsto nbh\}$
a7 : $pktSeqNo := \{pkt\} \lhd pktSeqNo$
a8 : $pktSrc := \{pkt\} \lhd pktSrc$
a9 : $pktFwdr := \{pkt\} \lhd pktFwdr$
a10 : $pktData := \{pkt\} \lhd pktData$
a11 : $pktNbHops := \{pkt\} \lhd pktNbHops$

end

Event *find_neighbours* $\widehat{=}$

//(env) channel finds the neighbour list of the forwarder

extends *find_neighbour*

any

s
nbs
f
pkt

where

g1 : $channel \neq \varnothing$
g2 : $pkt \in ran(channel)$
g3 : $s = initialSrcAddr(pkt)$
g4 : $f \mapsto pkt \in channel$
g5 : $nbs \subseteq ND \wedge nbs = wsnLinks[\{f\}] \wedge nbs \neq \varnothing$
g6 : $envNeighbours = \varnothing$

then

a1 : $envNeighbours := envNeighbours \cup (\{pkt\} \times nbs)$

end

Event *assign_forwarder* $\widehat{=}$

// (env) channel assigns specific forwarder - preparing for unicast for later refinement

extends *assign_forwarder*

any

s
nb
pkt
f
nbs

where

g1 : $channel \neq \varnothing$
g2 : $pkt \in ran(channel)$
g3 : $s = initialSrcAddr(pkt)$

g4 : $f \mapsto pkt \in channel$
g5 : $nbs = wsnLinks[\{f\}] \wedge nbs \neq \varnothing$
g6 : $nb \in nbs$

then

a1 : $envNeighbours := envNeighbours \cup \{pkt \mapsto nb\}$

end

Event *send_up* $\widehat{=}$

//(shd) channel sends the neighbour information

extends *send_up*

any

pkt
$nbrs$
f
sno
src
$fwdr$
$data$
nbh

where

g1 : $pkt \in ran(sentDown)$
g2 : $f \mapsto pkt \in sentDown$
g3 : $nbrs \in \{n | n \in (\mathbb{P}(ND)) \cup \{\{FAILED\_XMIT\}\}\}$
g4 : $nbrs \neq \varnothing$
g5 : $ran(\{pkt\} \lhd envNeighbours) \in (\mathbb{P}(ND) \cup \{\{FAILED\_XMIT\}\})$
g6 : $nbrs = ran(\{pkt\} \lhd envNeighbours) \wedge nbrs \neq \varnothing$
g7 : $pkt \notin dom(ctlNeighbours)$
g8 : $f \mapsto pkt \notin sentUp$
g9 : $f \mapsto pkt \in channel$
g10 : $pkt \in dom(vPktSeqNo) \wedge sno = vPktSeqNo(pkt)$
g11 : $pkt \in dom(vPktSrc) \wedge src = vPktSrc(pkt)$
g12 : $pkt \in dom(vPktFwdr) \wedge fwdr = vPktFwdr(pkt)$
g13 : $pkt \in dom(vPktData) \wedge data = vPktData(pkt)$
g14 : $pkt \in dom(vPktNbHops) \wedge nbh = vPktNbHops(pkt)$
g15 : $pkt \notin dom(pktSeqNo)$
g16 : $pkt \notin dom(pktSrc)$
g17 : $pkt \notin dom(pktFwdr)$
g18 : $pkt \notin dom(pktData)$
g19 : $pkt \notin dom(pktNbHops)$

then

a1 : $ctlNeighbours := ctlNeighbours \cup (\{pkt\} \times nbrs)$
a2 : $envNeighbours := \varnothing$

a3 : $channel := channel \setminus \{f \mapsto pkt\}$
a4 : $sentDown := sentDown \setminus \{f \mapsto pkt\}$
a5 : $sentUp := sentUp \cup \{f \mapsto pkt\}$
a6 : $pktSeqNo := pktSeqNo \cup \{pkt \mapsto sno\}$
a7 : $pktSrc := pktSrc \cup \{pkt \mapsto src\}$
a8 : $pktFwdr := pktFwdr \cup \{pkt \mapsto fwdr\}$
a9 : $pktData := pktData \cup \{pkt \mapsto data\}$
a10 : $pktNbHops := pktNbHops \cup \{pkt \mapsto nbh\}$
a11 : $vPktSeqNo := \{pkt\} \lhd vPktSeqNo$
a12 : $vPktSrc := \{pkt\} \lhd vPktSrc$
a13 : $vPktFwdr := \{pkt\} \lhd vPktFwdr$
a14 : $vPktData := \{pkt\} \lhd vPktData$
a15 : $vPktNbHops := \{pkt\} \lhd vPktNbHops$

end

Event *receive_dataPkt* $\widehat{=}$

//(ctl) each neighbour node receives a data packet

extends *receive_pkt*

any

s
nb
pkt
nbh

where

g1 : $pkt \notin ran(sentDown) \wedge pkt \in ran(sentUp)$
g2 : $pkt \mapsto nb \in ctlNeighbours$
g3 : $nb \mapsto pkt \notin ndBuff$
g4 : $nb \mapsto pkt \notin (sentUp \cup sentDown)$
g5 : $s = initialSrcAddr(pkt) \wedge s \neq nb$
g6 : $nb \neq Sink$
g7 : $nb \in dom(floodTbl) \wedge pkt \notin floodTbl(nb)$
g8 : $pkt \in dom(pktNbHops) \wedge nbh = pktNbHops(pkt) + 1$
g9 : $type(pkt) = DATA$

then

a1 : $ndBuff := ndBuff \cup \{nb \mapsto pkt\}$
a2 : $ctlNeighbours := ctlNeighbours \setminus \{pkt \mapsto nb\}$
a3 : $floodTbl(nb) := floodTbl(nb) \cup \{pkt\}$
a4 : $nbHops := nbHops \cup \{nb \mapsto \{pkt \mapsto nbh\}\}$

end

Event *receive_dup_dataPkt* $\widehat{=}$

//(ctl) each neighbour node discards a duplicated data packet

extends *receive_dup_pkt*

any

nb

pkt

where

g1 : $pkt \in dom(ctlNeighbours)$

g2 : $pkt \mapsto nb \in ctlNeighbours$

g3 : $nb \in dom(floodTbl) \wedge pkt \in floodTbl(nb)$

g3 : $type(pkt) = DATA$

then

a1 : $ctlNeighbours := ctlNeighbours \setminus \{pkt \mapsto nb\}$

end

Event *receive_controlPkt* $\widehat{=}$

//(ctl) each neighbour node receives a control packet

extends *receive_pkt*

any

s

nb

pkt

nbh

f

where

g1 : $pkt \notin ran(sentDown) \wedge pkt \in ran(sentUp)$

g2 : $pkt \mapsto nb \in ctlNeighbours$

g3 : $nb \mapsto pkt \notin ndBuff$

g4 : $nb \mapsto pkt \notin (sentUp \cup sentDown)$

g5 : $s = initialSrcAddr(pkt) \wedge s \neq nb$

g6 : $nb \neq Sink$

g7 : $nb \in dom(floodTbl) \wedge pkt \notin floodTbl(nb)$

g8 : $pkt \in dom(pktNbHops) \wedge nbh = pktNbHops(pkt) + 1$

g9 : $type(pkt) \in CONTROL$

g10 : $pkt \in dom(pktFwdr) \wedge f = pktFwdr(pkt)$

g11 : $f \in ND \wedge nb \in ND$

g12 : $f \mapsto nb \notin updateNbrs$

//nb has not been updated in the neighbour table yet

then

a1 : $ndBuff := ndBuff \cup \{nb \mapsto pkt\}$

a2 : $ctlNeighbours := ctlNeighbours \setminus \{pkt \mapsto nb\}$

a3 : $floodTbl(nb) := floodTbl(nb) \cup \{pkt\}$

a4 : $nbHops := nbHops \cup \{nb \mapsto \{pkt \mapsto nbh\}\}$

a5 : $updateNbrs := updateNbrs \cup \{f \mapsto nb\}$
//put nb into updateNbrs for updating it into the neighbour table later

end

Event *receive_dup_controlPkt* $\widehat{=}$

//(ctl) each neighbour node discards a duplicated control packet

extends *receive_dup_pkt*

any

nb
pkt
f

where

g1 : $pkt \in dom(ctlNeighbours)$
g2 : $pkt \mapsto nb \in ctlNeighbours$
g3 : $nb \in dom(floodTbl) \wedge pkt \in floodTbl(nb)$
g4 : $type(pkt) \in CONTROL$
g5 : $pkt \in dom(pktFwdr) \wedge f = pktFwdr(pkt)$
g6 : $f \in ND \wedge nb \in ND$
g7 : $f \mapsto nb \notin updateNbrs$
//nb has not been updated in the neighbour table yet

then

a1 : $ctlNeighbours := ctlNeighbours \setminus \{pkt \mapsto nb\}$
a2 : $updateNbrs := updateNbrs \cup \{f \mapsto nb\}$
//put nb into updateNbrs for updating it into the neighbour table later

end

Event *sink_recv_dataPkt* $\widehat{=}$

//(ctl) a sink receives a data packet from a channel

extends *sink_recv_pkt*

any

s
pkt
nb

where

g1 : $s = initialSrcAddr(pkt)$
g2 : $s \neq Sink$
g3 : $pkt \in ran(sentDown \cup sentUp) \setminus sinkBuff$
g4 : $nb = Sink$
g5 : $pkt \mapsto nb \in ctlNeighbours$
g6 : $nb \in dom(floodTbl) \wedge pkt \notin floodTbl(nb)$
g7 : $type(pkt) = DATA$

then

a1 : $sinkBuff := sinkBuff \cup \{pkt\}$
a2 : $ctlNeighbours := ctlNeighbours \setminus \{pkt \mapsto nb\}$
a3 : $floodTbl(nb) := floodTbl(nb) \cup \{pkt\}$

end

Event *sink_recv_controlPkt* $\widehat{=}$

//(ctl) a sink receives a control packet from a channel

extends *sink_recv_pkt*

any

s
pkt
nb
f

where

g1 : $s = initialSrcAddr(pkt)$
g2 : $s \neq Sink$
g3 : $pkt \in ran(sentDown \cup sentUp) \setminus sinkBuff$
g4 : $nb = Sink$
g5 : $pkt \mapsto nb \in ctlNeighbours$
g6 : $nb \in dom(floodTbl) \wedge pkt \notin floodTbl(nb)$
g7 : $type(pkt) \neq DATA$
g8 : $pkt \in dom(pktFwdr) \wedge f = pktFwdr(pkt)$
g9 : $f \in ND \wedge nb \in ND$
g10 : $f \mapsto nb \notin updateNbrs$

//nb has not been updated in the neighbour table yet

then

a1 : $sinkBuff := sinkBuff \cup \{pkt\}$
a2 : $ctlNeighbours := ctlNeighbours \setminus \{pkt \mapsto nb\}$
a3 : $floodTbl(nb) := floodTbl(nb) \cup \{pkt\}$
a4 : $updateNbrs := updateNbrs \cup \{f \mapsto nb\}$

//put nb into updateNbrs for updating it into the neighbour table later

end

Event *add_newEntry* $\widehat{=}$

//(ctl) adding new entry into the neighbour table

any

x
y

where

g1 : $updateNbrs \neq \varnothing$

//there are some nodes that have to update thier neighbour table

g2 : $x \in ND \wedge y \in ND \wedge x \mapsto y \in updateNbrs$
g3 : $y \mapsto x \notin neighbourTbl$

//in case these new pairs don't appear in the neighbour table

g4 : $y \mapsto x \mapsto 0 \notin missed$

g5 : $y \mapsto x \mapsto 0 \notin received$

g6 : $y \mapsto x \mapsto 0 \notin lastSeqno$

g7 : $y \mapsto x \mapsto 0 \notin receiveEst$

then

a1 : $neighbourTbl := neighbourTbl \cup \{y \mapsto x\}$

a2 : $missed := missed \cup \{y \mapsto x \mapsto 0\}$

a3 : $received := received \cup \{y \mapsto x \mapsto 0\}$

a4 : $lastSeqno := lastSeqno \cup \{y \mapsto x \mapsto 0\}$

a5 : $receiveEst := receiveEst \cup \{y \mapsto x \mapsto 0\}$

end

Event *update_nbr* $\widehat{=}$

//(ctl) updating received information into the neighbour table

any

x

y

sNo

$delta$

pkt

where

g1 : $updateNbrs \neq \varnothing$

//there are some nodes that have to update thier neighbour table

g2 : $x \in ND \wedge y \in ND \wedge x \mapsto y \in updateNbrs$

g3 : $y \mapsto x \in neighbourTbl$

// in case there are existing pairs in the neighbour table

g4 : $pkt \mapsto x \in pktFwdr$

g5 : $sNo = netSeqNo(pkt)$

//retriving the sequence number from the transmitted packet

g6 : $delta = sNo - lastSeqno(y \mapsto x) - 1$

//calculating the missed sequence number

g7 : $delta \geq 0$

g8 : $type(pkt) = BEACON$

g9 : $pkt \in ran(sentUp)$

then

a1 : $received := received \lhd\!\!\!- \{y \mapsto x \mapsto (received(y \mapsto x) + 1)\}$

a2 : $lastSeqno := lastSeqno \lhd\!\!\!- \{y \mapsto x \mapsto sNo\}$

a3 : $missed := missed \lhd\!\!\!- \{y \mapsto x \mapsto delta\}$

a4 : $updateNbrs := updateNbrs \setminus \{x \mapsto y\}$

end

Event *update_nbr2* $\widehat{=}$

//(ctl) preparing for update route packet for the later refinement

any

x

y

pkt

where

g1 : $updateNbrs \neq \varnothing$ //there are some nodes that have to process something

g2 : $x \in ND \wedge y \in ND \wedge x \mapsto y \in updateNbrs$

g3 : $y \mapsto x \in neighbourTbl$

g4 : $pkt \mapsto x \in pktFwdr$

g5 : $type(pkt) \neq DATA \wedge type(pkt) \neq BEACON$

//but not from neither data or beacon packet

g6 : $pkt \in ran(sentUp)$

then

a1 : $updateNbrs := updateNbrs \setminus \{x \mapsto y\}$

end

Event *est_due* $\widehat{=}$

//(ctl) due the link quality estimation

any

x

where

g1 : $x \in ND \setminus estNDs$ //the node that has not calculated quality estimation yet

g2 : $x \in dom(totalSentBcon) \wedge totalSentBcon(x) > 0$

g3 : $totalSentBcon(x) mod EST\_RATIO = 0$

//the number of transmitted packet reaches the estimation ratio

then

a1 : $estNDs := estNDs \cup \{x\}$

//updating node x in estNDs for considering to update reception ratio

end

Event *update_est* $\widehat{=}$

//(cal) calculating and updating reception ratio in case no lost packets or lose packets in the middle of transmission

any

nd

nb

r

m

rt

$newAve$

where

g1 : $nd \in estNDs$ //the link quality estimation period of node nd must be due

g2 : $nb \in ran(\{nd\} \lhd neighbourTbl)$

//finding the neighbours of node nd in the neighbour table

g3 : $nd \in dom(totalSentBcon) \wedge totalSentBcon(nd) > 0$

//node nd should transmit at least one packet

g4 : $nd \mapsto nb \in dom(lastSeqno) \wedge lastSeqno(nd \mapsto nb) > 0$

g5 : $r = received(nd \mapsto nb)$

g6 : $m = missed(nd \mapsto nb)$

g7 : $rt = EST\_RATIO$

g8 : $newAve = ((r * 100)/rt)$ //calculating the reception ratio

g9 : $nd \mapsto nb \in dom(receiveEst) \wedge newAve > 0$

g10 : $nd \mapsto nb \in dom(missed)$

g11 : $nd \mapsto nb \in dom(received)$

g12 : $nd \mapsto nb \notin estNbrs$

g13 : $estNbrs \neq neighbourTbl$

then

a1 : $receiveEst := receiveEst \Leftarrow \{nd \mapsto nb \mapsto newAve\}$

a2 : $missed := missed \Leftarrow \{nd \mapsto nb \mapsto 0\}$

a3 : $received := received \Leftarrow \{nd \mapsto nb \mapsto 0\}$

a4 : $estNbrs := estNbrs \cup \{nd \mapsto nb\}$

end

Event *update_est_nothing* $\widehat{=}$

//(ctl) for case the dead neighbour node affecting received become zero

any

nd

nb

r

m

rt

$newAve$

where

g1 : $nd \in estNDs$

g2 : $nb \in ran(\{nd\} \lhd neighbourTbl)$

g3 : $nd \in dom(totalSentBcon) \wedge totalSentBcon(nd) > 0$

//node nd should transmit at least one packet

g4 : $nd \mapsto nb \in dom(lastSeqno) \wedge lastSeqno(nd \mapsto nb) > 0$

g5 : $r = received(nd \mapsto nb)$

g6 : $m = missed(nd \mapsto nb)$

g7 : $rt = EST\_RATIO \wedge rt > 0$

g8 : $newAve = ((r * 100)/rt)$ //calculating the reception ratio

g9 : $newAve = 0$

g10 : $nd \mapsto nb \notin estNbrs$

g11 : $estNbrs \neq neighbourTbl$

then

a1 : $estNbrs := estNbrs \cup \{nd \mapsto nb\}$

end

Event *reset_estTimer* $\widehat{=}$

//(ctl) resetting estimation timer for the next round

when

g1 : $estNDs = ND$

g2 : $totalSentBcon \neq \varnothing$

g3 : $estNbrs = neighbourTbl$

then

a1 : $estNDs := \varnothing$

a3 : $estNbrs := \varnothing$

a4 : $totalSentBcon := ND \times \{0\}$

end

Event *finish_tx_pkt* $\widehat{=}$

//(ctl) finishing packet transmission

extends *finish_tx_pkt*

any

f

pkt

where

g1 : $\{pkt\} \lhd ctlNeighbours = \varnothing$

g2 : $pkt \in ran(sentUp) \wedge pkt \notin ran(sentDown) \wedge pkt \in ran(ndBuff)$

g3 : $f \in dom(sentUp) \wedge f \mapsto pkt \in sentUp$

then

a1 : $sentUp := sentUp \setminus \{f \mapsto pkt\}$

end

Event *final_tx_pkt* $\widehat{=}$

//(ctl) the transmitted packet is removed from middleware indicating completion of transmission mechanism

extends *final_tx_pkt*

any

pkt

where

g1 : $\{pkt\} \lhd ctlNeighbours = \varnothing$

g2 : $pkt \in ran(sentUp) \wedge pkt \notin ran(sentDown) \wedge pkt \notin ran(ndBuff)$

then

a1 : $sentUp := sentUp \rhd \{pkt\}$

end

Event *lose_all_neighbours* $\widehat{=}$

//(env) channel loses all neighbuor

extends *lose_all_neighbours*

any

s

nbs

f

pkt

where

g1 : $channel \neq \varnothing$

g2 : $pkt \in ran(channel)$

g3 : $s = initialSrcAddr(pkt)$

g4 : $f \mapsto pkt \in channel$

g5 : $nbs = wsnLinks[\{f\}] \wedge nbs = \varnothing$

then

a1 : $envNeighbours := envNeighbours \cup \{pkt \mapsto FAILED\_XMIT\}$

end

Event *lose_pkt* $\widehat{=}$

//(ctl) each node records a lost packet

extends *lose_pkt*

any

f

pkt

where

g1 : $pkt \in ran(sentUp) \wedge pkt \notin ran(sentDown) \wedge pkt \notin ran(ndBuff)$

g2 : $pkt \notin ran(recLostPkts)$

g3 : $f \mapsto pkt \in sentUp \wedge f \mapsto pkt \notin sentDown$

g4 : $f \mapsto pkt \notin recLostPkts$

g5 : $ran(\{pkt\} \lhd ctlNeighbours) = \{FAILED\_XMIT\}$

then

a1 : $sentUp := sentUp \rhd\!\!\!- \{pkt\}$

a2 : $recLostPkts := recLostPkts \cup \{f \mapsto pkt\}$

end

END

A.5 Fourth refinement: route broadcast mechanism

A.5.1 Machine:M4

MACHINE M4

REFINES M3

SEES C3

VARIABLES

$sinkBuff$ //(ctl) a sink buffer

$ndBuff$ //(ctl) nodes' waiting buffer

$sensingNDs$ //(ctl) sensing nodes

$sensedPkts$ //(ctl) sensed packets
$senseBuff$ //(ctl) sense buffer
$floodedPkts$ //(ctl) flooded packets
$floodTbl$ //(ctl)historical received packets used for detect duplicated packets
$ctlNeighbours$ //(ctl) neighbour lists
$recLostPkts$ //(ctl) recorded lost packet at each node
$envSensedFlg$ //(env) sensing period
$envData$ //(env) environmental/conditional data
$ctlSensedFlg$ //(ctl) sensing period
$sentDown$ //(ctl) transmitted packets down to channel
$sentUp$ //(ctl) transmitted packets up from channel
$channel$ //(env) channel
$envNeighbours$ //(env) neighbour list
$wsnLinks$ //(env) dynamic WSN links
$floodFlg$ //(ctl) flooded flag
$dataSeqNo$ //(ctl) local nodes' current data sequence number at application layer
$floodSeqNo$ //(ctl) local nodes' current broadcast sequence number at application layer
$nbHops$ //(ctl) neighbour's hop count
$crashedLinks$ //(env) crashed WSN Links
$pktSrc$ //(ctl) src address attached in transmitted packet
$pktSeqNo$ //(ctl) sequence number attached in transmitted packet
$pktFwdr$ //(ctl) fowarder attached in transmitted packet
$pktData$ //(ctl) data attached in transmitted packet
$pktNbHops$ //(ctl) hop count attached in transmitted packet
$vPktSeqNo$ //(env) virtual packet at channel
$vPktSrc$ //(env) virtual packet at channel
$vPktFwdr$ //(env) virtual packet at channel
$vPktData$ //(env) virtual packet at channel
$vPktNbHops$ //(env) virtual packet at channel
$totalSentBcon$ //(ctl) number of sent beacon packets, used for cost-estimation
$netSeqNo$ //(ctl) link sequence no in a packet, used for cost-estimation
$linkSeqNo$ //(ctl) link sequence number of each node
$updateNbrs$ //(ctl) for updating neighbours in the neighbour table
$neighbourTbl$ //(ctl)neighbour table structure
$missed$ //(ctl) neighbour table: field missed (number of lost packets)
$received$ //(ctl) neighbour table: field received (number of received packets)
$lastSeqno$ //(ctl) neighbour table: field lastSeqno in neighbour table
$receiveEst$ //(ctl) neighbour talble: field receiveEst (reception link quality)
$estNDs$ //(ctl) estimated nodes
$estNbrs$ //(ctl) estimated neighbours

$bcastRouTimer$ //(ctl) broadcast route timer
$bcastRouNodes$ //(ctl) route broadcast nodes
$routeSeqNo$ //(ctl) sequence number for route packet at each node
$sentEst$ //(ctl) neighbour table: field sentEst (transmission quality) in neighbour table
$liveliness$ //(ctl) neighbour table: field liveliness in neighbour table
$checkedLiveNbrs$ //(ctl) checked liveliness nodes

INVARIANTS

inv4_1 : $sentEst \in neighbourTbl \rightarrow \mathbb{N}$
inv4_2 : $bcastRouTimer \in BOOL$
inv4_3 : $bcastRouNodes \subseteq ND$
inv4_4 : $routeSeqNo \in ND \rightarrow \mathbb{N}$
inv4_5 : $liveliness \in neighbourTbl \rightarrow \mathbb{N}$
inv4_6 : $checkedLiveNbrs \subseteq neighbourTbl$

EVENTS

Initialisation

extended

begin

int0_3 : $sinkBuff := \varnothing$
int1_1 : $ndBuff := \varnothing$
int1_3 : $sensingNDs := \varnothing$
int1_4 : $sensedPkts := \varnothing$
int1_5 : $senseBuff := (ND \setminus \{Sink\}) \times \{\varnothing\}$
int1_6 : $floodedPkts := \varnothing$
int1_7 : $floodTbl := ND \times \{\varnothing\}$
int1_8 : $ctlNeighbours := \varnothing$
int1_9 : $envSensedFlg := (ND \setminus \{Sink\}) \times \{FALSE\}$
int1_10 : $envData := (ND \setminus \{Sink\}) \times \{0\}$
int1_11 : $ctlSensedFlg := (ND \setminus \{Sink\}) \times \{FALSE\}$
int1_12 : $floodFlg := ND \times \{FALSE\}$
inv1_20 : $recLostPkts := \varnothing$
int2_1 : $sentUp := \varnothing$
int2_2 : $sentDown := \varnothing$
int2_3 : $channel := \varnothing$
int2_10 : $envNeighbours := \varnothing$
int2_11 : $wsnLinks := \varnothing$
int2_12 : $crashedLinks := \varnothing$
int2_13 : $dataSeqNo := ND \times \{0\}$
int2_14 : $floodSeqNo := ND \times \{0\}$
int2_15 : $nbHops := \varnothing$
int2_16 : $pktSeqNo := \varnothing$
int2_17 : $pktSrc := \varnothing$

int2_18 : $pktFwdr := \varnothing$
int2_19 : $pktData := \varnothing$
int2_20 : $pktNbHops := \varnothing$
int2_21 : $vPktSeqNo := \varnothing$
int2_22 : $vPktSrc := \varnothing$
int2_23 : $vPktFwdr := \varnothing$
int2_24 : $vPktData := \varnothing$
int2_25 : $vPktNbHops := \varnothing$
int3_1 : $totalSentBcon := ND \times \{0\}$
int3_2 : $netSeqNo := PKT \times \{0\}$
int3_3 : $updateNbrs := \varnothing$
int3_4 : $neighbourTbl := \varnothing$
int3_5 : $missed := \varnothing$
int3_6 : $received := \varnothing$
int3_7 : $lastSeqno := \varnothing$
int3_8 : $receiveEst := \varnothing$
int3_9 : $linkSeqNo := ND \times \{0\}$
int3_11 : $estNDs := \varnothing$
int3_12 : $estNbrs := \varnothing$
int4_1 : $sentEst := \varnothing$
int4_2 : $bcastRouTimer := FALSE$
int4_3 : $bcastRouNodes := \varnothing$
int4_4 : $routeSeqNo := ND \times \{0\}$
int4_5 : $liveliness := \varnothing$
int4_6 : $checkedLiveNbrs := \varnothing$

end

Event *add_link* $\widehat{=}$

//(env) channel sets a topology

extends *add_link*

any

x

y

where

g1 : $x \in ND \wedge y \in ND$
g2 : $x \mapsto y \notin wsnLinks$
g3 : $x \neq y$

then

a1 : $wsnLinks := wsnLinks \cup \{x \mapsto y\}$

end

Event *disconnect_link* $\widehat{=}$

//(env) disconnecting a link

extends *disconnect_link*

any

x

y

where

g1 : $x \in ND \wedge y \in ND$

g2 : $x \mapsto y \in wsnLinks$

g3 : $x \mapsto y \notin crashedLinks$

then

a1 : $crashedLinks := crashedLinks \cup \{x \mapsto y\}$

a2 : $wsnLinks := wsnLinks \setminus \{x \mapsto y\}$

end

Event *recover_link* $\widehat{=}$

//(env) recovering a link

extends *recover_link*

any

x

y

where

g1 : $x \in ND \wedge y \in ND$

g2 : $x \mapsto y \in crashedLinks$

g3 : $x \mapsto y \notin wsnLinks$

g4 : $x \neq y$

then

a1 : $wsnLinks := wsnLinks \cup \{x \mapsto y\}$

a2 : $crashedLinks := crashedLinks \setminus \{x \mapsto y\}$

end

Event *start_sensing* $\widehat{=}$

//(env) activating each node to sense data

extends *start_sensing*

any

x

sd

where

g1 : $x \in ND \setminus \{Sink\}$

g2 : $envSensedFlg(x) = FALSE$

g3 : $sd \in randomFn$

then

a1 : $envSensedFlg(x) := TRUE$

a2 : $envData(x) := sd$

end

Event *sensing* $\widehat{=}$

//(shd) each node senses a data from environment

extends *sensing*

any

x

sf

sd

where

g1 : $x \in ND \setminus \{Sink\}$

g2 : $sf = envSensedFlg(x)$

g3 : $envSensedFlg(x) = TRUE$

g4 : $ctlSensedFlg(x) = FALSE$

g5 : $sd = envData(x)$

then

a1 : $ctlSensedFlg(x) := sf$

a2 : $senseBuff(x) := senseBuff(x) \cup \{sd\}$

end

Event *finish_sensing* $\widehat{=}$

//(shd) finishing sensing data process

extends *finish_sensing*

any

x

where

g1 : $x \in dom(envSensedFlg)$

g2 : $envSensedFlg(x) = TRUE$

g3 : $ndBuff = \varnothing \wedge (sentDown \cup sentUp) = \varnothing$

g4 : $ctlSensedFlg(x) = TRUE$

g5 : $sensingNDs = \{x\}$

then

a1 : $ctlSensedFlg(x) := FALSE$

a2 : $envSensedFlg(x) := FALSE$

a3 : $sensingNDs := \varnothing$

end

Event *create_dataPkt* $\widehat{=}$

//(ctl) creating a data packet

extends *create_dataPkt*

any

s

pkt

sd
sno
nbh

where

g1 : $pkt \in PKT \wedge s \in ND \setminus \{Sink\}$
g2 : $pkt \notin ran(ndBuff \cup (sentUp \cup sentDown))$
g3 : $s \notin sensingNDs \wedge pkt \notin sensedPkts \cup floodedPkts$
g4 : $s = initialSrcAddr(pkt)$
g5 : $s \in dom(floodTbl) \wedge pkt \notin floodTbl(s)$
g6 : $type(pkt) = DATA$
g7 : $ctlSensedFlg(s) = TRUE$
g8 : $senseBuff \neq \varnothing \wedge sd \in senseBuff(s)$
g9 : $sno = dataSeqNo(s) + 1 \wedge s \in dom(dataSeqNo)$
g10 : $nbh \in \mathbb{Z} \wedge nbh = -1$
g11 : $s \mapsto \{pkt \mapsto nbh\} \notin nbHops$
g12 : $pkt \notin dom(pktSeqNo)$
g13 : $pkt \notin dom(pktSrc)$
g14 : $pkt \notin dom(pktFwdr)$
g15 : $pkt \notin dom(pktData)$
g16 : $pkt \notin dom(pktNbHops)$

then

a1 : $sensingNDs := sensingNDs \cup \{s\}$
a2 : $sensedPkts := sensedPkts \cup \{pkt\}$
a3 : $ndBuff := ndBuff \cup \{s \mapsto pkt\}$
a4 : $floodTbl(s) := floodTbl(s) \cup \{pkt\}$
a5 : $dataSeqNo := dataSeqNo \lhd\!\!\!- \{s \mapsto sno\}$
a6 : $nbHops := nbHops \cup \{s \mapsto \{pkt \mapsto nbh\}\}$
a7 : $senseBuff(s) := senseBuff(s) \setminus \{sd\}$
a8 : $pktSeqNo := pktSeqNo \lhd\!\!\!- \{pkt \mapsto sno\}$
a9 : $pktSrc := pktSrc \cup \{pkt \mapsto s\}$
a10 : $pktFwdr := pktFwdr \cup \{pkt \mapsto s\}$
a11 : $pktData := pktData \cup \{pkt \mapsto sd\}$
a12 : $pktNbHops := pktNbHops \cup \{pkt \mapsto nbh\}$

end

Event *create_bconPkt* $\widehat{=}$

//(ctl) creating a beacon packet

extends *create_controlPkt*

any

s
pkt
sno

nbh

where

g1 : $pkt \in PKT$

g2 : $pkt \notin ran(ndBuff \cup (sentUp \cup sentDown))$

g3 : $pkt \notin (sensedPkts \cup floodedPkts)$

g4 : $s = initialSrcAddr(pkt)$

g5 : $s \in dom(floodTbl) \wedge pkt \notin floodTbl(s)$

g6 : $type(pkt) \in CONTROL$

g7 : $floodFlg(s) = TRUE$

g8 : $sno = floodSeqNo(s) + 1 \wedge s \in dom(floodSeqNo)$

g9 : $nbh \in \mathbb{Z} \wedge nbh = -1$

g10 : $s \mapsto \{pkt \mapsto nbh\} \notin nbHops$

g11 : $pkt \notin dom(pktSeqNo)$

g12 : $pkt \notin dom(pktSrc)$

g13 : $pkt \notin dom(pktFwdr)$

g14 : $pkt \notin dom(pktData)$

g15 : $pkt \notin dom(pktNbHops)$

g16 : $s = Sink$

g17 : $type(pkt) = BEACON$

then

a1 : $floodedPkts := floodedPkts \cup \{pkt\}$

a2 : $ndBuff := ndBuff \cup \{s \mapsto pkt\}$

a3 : $floodTbl(s) := floodTbl(s) \cup \{pkt\}$

a4 : $floodSeqNo := floodSeqNo \lhd\!\!\!- \{s \mapsto sno\}$

a5 : $nbHops := nbHops \cup \{s \mapsto \{pkt \mapsto nbh\}\}$

a6 : $pktSeqNo := pktSeqNo \lhd\!\!\!- \{pkt \mapsto sno\}$

a7 : $pktFwdr := pktFwdr \cup \{pkt \mapsto s\}$

a8 : $pktSrc := pktSrc \cup \{pkt \mapsto s\}$

a9 : $pktData := pktData \cup \{pkt \mapsto 0\}$

a10 : $pktNbHops := pktNbHops \cup \{pkt \mapsto nbh\}$

end

Event *bcastRou_due* $\widehat{=}$

//(ctl) broadcast route timer due

extends *reset_estTimer*

when

g1 : $estNDs = ND$

g2 : $totalSentBcon \neq \varnothing$

g3 : $estNbrs = neighbourTbl$

g4 : $bcastRouTimer = FALSE$

g5 : $checkedLiveNbrs = \varnothing$

then

a1 : $estNDs := \varnothing$

a3 : $estNbrs := \varnothing$

a4 : $totalSentBcon := ND \times \{0\}$

a5 : $bcastRouTimer := TRUE$

end

Event *reset_bcastRouTimer* $\widehat{=}$

//(ctl) resetting broadcast route timer for the next round

when

g1 : $bcastRouTimer = TRUE$

g2 : $bcastRouNodes = ND$
//when all node broadcast the route packet

then

a1 : $bcastRouTimer := FALSE$

a2 : $bcastRouNodes := \varnothing$

end

Event *create_routePkt* $\widehat{=}$

//(ctl) creating a route packet

extends *create_controlPkt*

any

s

pkt

sno

nbh

where

g1 : $pkt \in PKT$

g2 : $pkt \notin ran(ndBuff \cup (sentUp \cup sentDown))$

g3 : $pkt \notin (sensedPkts \cup floodedPkts)$

g4 : $s = initialSrcAddr(pkt)$

g5 : $s \in dom(floodTbl) \wedge pkt \notin floodTbl(s)$

g6 : $type(pkt) \in CONTROL$

g7 : $floodFlg(s) = TRUE$

g8 : $sno = floodSeqNo(s) + 1 \wedge s \in dom(floodSeqNo)$

g9 : $nbh \in \mathbb{Z} \wedge nbh = -1$

g10 : $s \mapsto \{pkt \mapsto nbh\} \notin nbHops$

g11 : $pkt \notin dom(pktSeqNo)$

g12 : $pkt \notin dom(pktSrc)$

g13 : $pkt \notin dom(pktFwdr)$

g14 : $pkt \notin dom(pktData)$

g15 : $pkt \notin dom(pktNbHops)$

g16 : $type(pkt) = ROUTE$

g17 : $bcastRouTimer = TRUE$

g18 : $s \notin bcastRouNodes$

then

a1 : $floodedPkts := floodedPkts \cup \{pkt\}$

a2 : $ndBuff := ndBuff \cup \{s \mapsto pkt\}$

a3 : $floodTbl(s) := floodTbl(s) \cup \{pkt\}$

a4 : $floodSeqNo := floodSeqNo \lhd\!\!\!- \{s \mapsto sno\}$

a5 : $nbHops := nbHops \cup \{s \mapsto \{pkt \mapsto nbh\}\}$

a6 : $pktSeqNo := pktSeqNo \lhd\!\!\!- \{pkt \mapsto sno\}$

a7 : $pktFwdr := pktFwdr \cup \{pkt \mapsto s\}$

a8 : $pktSrc := pktSrc \cup \{pkt \mapsto s\}$

a9 : $pktData := pktData \cup \{pkt \mapsto 0\}$

a10 : $pktNbHops := pktNbHops \cup \{pkt \mapsto nbh\}$

a12 : $bcastRouNodes := bcastRouNodes \cup \{s\}$

end

Event *start_tx_dataPkt* $\widehat{=}$

//(ctl) starting transmiting a data packet to a channel

extends *start_tx_dataPkt*

any

x

pkt

nbh

where

g1 : $x \mapsto pkt \in ndBuff$

g2 : $x \mapsto pkt \notin sentDown$

g3 : $x \mapsto pkt \notin sentUp$

g4 : $pkt \in dom(pktFwdr)$

g5 : $x \mapsto \{pkt \mapsto nbh\} \in nbHops$

g6 : $pkt \in dom(pktNbHops)$

g7 : $type(pkt) = DATA$

then

a1 : $ndBuff := ndBuff \setminus \{x \mapsto pkt\}$

a2 : $sentDown := sentDown \cup \{x \mapsto pkt\}$

a3 : $pktFwdr := pktFwdr \lhd\!\!\!- \{pkt \mapsto x\}$

a4 : $pktNbHops := pktNbHops \lhd\!\!\!- \{pkt \mapsto nbh\}$

a5 : $nbHops := nbHops \setminus \{x \mapsto \{pkt \mapsto nbh\}\}$

end

Event *start_tx_bconPkt* $\widehat{=}$

//(ctl) starting transmitting a beacon packet to a channel

extends *start_tx_bconPkt*

any

x

pkt

nbh

$lsno$

where

g1 : $x \mapsto pkt \in ndBuff$

g2 : $x \mapsto pkt \notin sentDown$

g3 : $x \mapsto pkt \notin sentUp$

g4 : $pkt \in dom(pktFwdr)$

g5 : $x \mapsto \{pkt \mapsto nbh\} \in nbHops$

g6 : $pkt \in dom(pktNbHops)$

g7 : $type(pkt) = BEACON$

g8 : $x \in dom(totalSentBcon) \wedge totalSentBcon(x) \geq 0$

g9 : $lsno = linkSeqNo(x) + 1 \wedge x \in dom(linkSeqNo)$

then

a1 : $ndBuff := ndBuff \setminus \{x \mapsto pkt\}$

a2 : $sentDown := sentDown \cup \{x \mapsto pkt\}$

a3 : $pktFwdr := pktFwdr \lhd\!\!\!- \{pkt \mapsto x\}$

a4 : $pktNbHops := pktNbHops \lhd\!\!\!- \{pkt \mapsto nbh\}$

a5 : $nbHops := nbHops \setminus \{x \mapsto \{pkt \mapsto nbh\}\}$

a6 : $totalSentBcon(x) := totalSentBcon(x) + 1$

a7 : $linkSeqNo := linkSeqNo \lhd\!\!\!- \{x \mapsto lsno\}$

a8 : $netSeqNo(pkt) := lsno$

end

Event *start_tx_routePkt* $\widehat{=}$

//(ctl) starting transmitting a route packet to a channel

extends *start_tx_controlPkt*

any

x

pkt

nbh

$lsno$

where

g1 : $x \mapsto pkt \in ndBuff$

g2 : $x \mapsto pkt \notin sentDown$

g3 : $x \mapsto pkt \notin sentUp$

g4 : $pkt \in dom(pktFwdr)$

g5 : $x \mapsto \{pkt \mapsto nbh\} \in nbHops$

g6 : $pkt \in dom(pktNbHops)$

g7 : $type(pkt) \neq BEACON$

g8 : $lsno \in \mathbb{N}$

g9 : $type(pkt) = ROUTE$

g10 : $x \in dom(routeSeqNo) \wedge lsno = routeSeqNo(x) + 1$

then

a1 : $ndBuff := ndBuff \setminus \{x \mapsto pkt\}$

a2 : $sentDown := sentDown \cup \{x \mapsto pkt\}$

a3 : $pktFwdr := pktFwdr \lhd\!\!\!- \{pkt \mapsto x\}$

a4 : $pktNbHops := pktNbHops \lhd\!\!\!- \{pkt \mapsto nbh\}$

a5 : $nbHops := nbHops \setminus \{x \mapsto \{pkt \mapsto nbh\}\}$

a6 : $netSeqNo(pkt) := lsno$

a7 : $routeSeqNo := routeSeqNo \lhd\!\!\!- \{x \mapsto lsno\}$

end

Event *send_down* $\widehat{=}$

//(shd) sending a packet down to a channel

extends *send_down*

any

cn

pkt

sno

src

$data$

$fwdr$

nbh

where

g1 : $cn \mapsto pkt \in sentDown$

g2 : $cn \mapsto pkt \notin channel$

g3 : $pkt \in dom(pktSeqNo) \wedge sno = pktSeqNo(pkt)$

g4 : $pkt \in dom(pktSrc) \wedge src = pktSrc(pkt)$

g5 : $pkt \in dom(pktFwdr) \wedge fwdr = pktFwdr(pkt)$

g6 : $pkt \in dom(pktData) \wedge data = pktData(pkt)$

g7 : $pkt \in dom(pktNbHops) \wedge nbh = pktNbHops(pkt)$

g8 : $pkt \notin dom(vPktSeqNo)$

g9 : $pkt \notin dom(vPktFwdr)$

g10 : $pkt \notin dom(vPktSrc)$

g11 : $pkt \notin dom(vPktData)$

g12 : $pkt \notin dom(vPktNbHops)$

then

a1 : $channel := channel \cup \{cn \mapsto pkt\}$

a2 : $vPktSeqNo := vPktSeqNo \cup \{pkt \mapsto sno\}$

a3 : $vPktSrc := vPktSrc \cup \{pkt \mapsto src\}$

a4 : $vPktFwdr := vPktFwdr \cup \{pkt \mapsto fwdr\}$

a5 : $vPktData := vPktData \cup \{pkt \mapsto data\}$

a6 : $vPktNbHops := vPktNbHops \cup \{pkt \mapsto nbh\}$

a7 : $pktSeqNo := \{pkt\} \lhd pktSeqNo$
a8 : $pktSrc := \{pkt\} \lhd pktSrc$
a9 : $pktFwdr := \{pkt\} \lhd pktFwdr$
a10 : $pktData := \{pkt\} \lhd pktData$
a11 : $pktNbHops := \{pkt\} \lhd pktNbHops$

end

Event *find_neighbours* $\widehat{=}$

//(env) channel finds the neighbour list of the forwarder

extends *find_neighbour*

any

s
nbs
f
pkt

where

g1 : $channel \neq \varnothing$
g2 : $pkt \in ran(channel)$
g3 : $s = initialSrcAddr(pkt)$
g4 : $f \mapsto pkt \in channel$
g5 : $nbs \subseteq ND \wedge nbs = wsnLinks[\{f\}] \wedge nbs \neq \varnothing$
g6 : $envNeighbours = \varnothing$

then

a1 : $envNeighbours := envNeighbours \cup (\{pkt\} \times nbs)$

end

Event *assign_forwarder* $\widehat{=}$

//(env) channel assigns specific forwarder - preparing for unicast for later refinement

extends *assign_forwarder*

any

s
nb
pkt
f
nbs

where

g1 : $channel \neq \varnothing$
g2 : $pkt \in ran(channel)$
g3 : $s = initialSrcAddr(pkt)$
g4 : $f \mapsto pkt \in channel$
g5 : $nbs = wsnLinks[\{f\}] \wedge nbs \neq \varnothing$

g6 : $nb \in nbs$

then

a1 : $envNeighbours := envNeighbours \cup \{pkt \mapsto nb\}$

end

Event *send_up* $\widehat{=}$

//(shd) channel sends the neighbour information

extends *send_up*

any

pkt
$nbrs$
f
sno
src
$fwdr$
$data$
nbh

where

g1 : $pkt \in ran(sentDown)$
g2 : $f \mapsto pkt \in sentDown$
g3 : $nbrs \in \{n | n \in (\mathbb{P}(ND)) \cup \{\{FAILED\_XMIT\}\}\}$
g4 : $nbrs \neq \varnothing$
g5 : $ran(\{pkt\} \lhd envNeighbours) \in (\mathbb{P}(ND) \cup \{\{FAILED\_XMIT\}\})$
g6 : $nbrs = ran(\{pkt\} \lhd envNeighbours) \wedge nbrs \neq \varnothing$
g7 : $pkt \notin dom(ctlNeighbours)$
g8 : $f \mapsto pkt \notin sentUp$
g9 : $f \mapsto pkt \in channel$
g10 : $pkt \in dom(vPktSeqNo) \wedge sno = vPktSeqNo(pkt)$
g11 : $pkt \in dom(vPktSrc) \wedge src = vPktSrc(pkt)$
g12 : $pkt \in dom(vPktFwdr) \wedge fwdr = vPktFwdr(pkt)$
g13 : $pkt \in dom(vPktData) \wedge data = vPktData(pkt)$
g14 : $pkt \in dom(vPktNbHops) \wedge nbh = vPktNbHops(pkt)$
g15 : $pkt \notin dom(pktSeqNo)$
g16 : $pkt \notin dom(pktSrc)$
g17 : $pkt \notin dom(pktFwdr)$
g18 : $pkt \notin dom(pktData)$
g19 : $pkt \notin dom(pktNbHops)$

then

a1 : $ctlNeighbours := ctlNeighbours \cup (\{pkt\} \times nbrs)$
a2 : $envNeighbours := \varnothing$
a3 : $channel := channel \setminus \{f \mapsto pkt\}$
a4 : $sentDown := sentDown \setminus \{f \mapsto pkt\}$

a5 : $sentUp := sentUp \cup \{f \mapsto pkt\}$
a6 : $pktSeqNo := pktSeqNo \cup \{pkt \mapsto sno\}$
a7 : $pktSrc := pktSrc \cup \{pkt \mapsto src\}$
a8 : $pktFwdr := pktFwdr \cup \{pkt \mapsto fwdr\}$
a9 : $pktData := pktData \cup \{pkt \mapsto data\}$
a10 : $pktNbHops := pktNbHops \cup \{pkt \mapsto nbh\}$
a11 : $vPktSeqNo := \{pkt\} \lhd vPktSeqNo$
a12 : $vPktSrc := \{pkt\} \lhd vPktSrc$
a13 : $vPktFwdr := \{pkt\} \lhd vPktFwdr$
a14 : $vPktData := \{pkt\} \lhd vPktData$
a15 : $vPktNbHops := \{pkt\} \lhd vPktNbHops$

end

Event *receive_dataPkt* $\widehat{=}$

//(ctl) each neighbour node receives a data packet from a channel

extends *receive_dataPkt*

any

s
nb
pkt
nbh

where

g1 : $pkt \notin ran(sentDown) \wedge pkt \in ran(sentUp)$
g2 : $pkt \mapsto nb \in ctlNeighbours$
g3 : $nb \mapsto pkt \notin ndBuff$
g4 : $nb \mapsto pkt \notin (sentUp \cup sentDown)$
g5 : $s = initialSrcAddr(pkt) \wedge s \neq nb$
g6 : $nb \neq Sink$
g7 : $nb \in dom(floodTbl) \wedge pkt \notin floodTbl(nb)$
g8 : $pkt \in dom(pktNbHops) \wedge nbh = pktNbHops(pkt) + 1$
g9 : $type(pkt) = DATA$

then

a1 : $ndBuff := ndBuff \cup \{nb \mapsto pkt\}$
a2 : $ctlNeighbours := ctlNeighbours \setminus \{pkt \mapsto nb\}$
a3 : $floodTbl(nb) := floodTbl(nb) \cup \{pkt\}$
a4 : $nbHops := nbHops \cup \{nb \mapsto \{pkt \mapsto nbh\}\}$

end

Event *receive_dup_dataPkt* $\widehat{=}$

//(ctl) each neighbour node discards a duplicated data packet

extends *receive_dup_dataPkt*

any

nb

pkt

where

g1 : $pkt \in dom(ctlNeighbours)$

g2 : $pkt \mapsto nb \in ctlNeighbours$

g3 : $nb \in dom(floodTbl) \wedge pkt \in floodTbl(nb)$

g4 : $type(pkt) = DATA$

then

a1 : $ctlNeighbours := ctlNeighbours \setminus \{pkt \mapsto nb\}$

end

Event *receive_controlPkt* $\widehat{=}$

//(ctl) each neighbour node receives a control packet

extends *receive_controlPkt*

any

s

nb

pkt

nbh

f

where

g1 : $pkt \notin ran(sentDown) \wedge pkt \in ran(sentUp)$

g2 : $pkt \mapsto nb \in ctlNeighbours$

g3 : $nb \mapsto pkt \notin ndBuff$

g4 : $nb \mapsto pkt \notin (sentUp \cup sentDown)$

g5 : $s = initialSrcAddr(pkt) \wedge s \neq nb$

g6 : $nb \neq Sink$

g7 : $nb \in dom(floodTbl) \wedge pkt \notin floodTbl(nb)$

g8 : $pkt \in dom(pktNbHops) \wedge nbh = pktNbHops(pkt) + 1$

g9 : $type(pkt) \in CONTROL$

g10 : $pkt \in dom(pktFwdr) \wedge f = pktFwdr(pkt)$

g11 : $f \in ND \wedge nb \in ND$

g12 : $f \mapsto nb \notin updateNbrs$

then

a1 : $ndBuff := ndBuff \cup \{nb \mapsto pkt\}$

a3 : $ctlNeighbours := ctlNeighbours \setminus \{pkt \mapsto nb\}$

a4 : $floodTbl(nb) := floodTbl(nb) \cup \{pkt\}$

a5 : $nbHops := nbHops \cup \{nb \mapsto \{pkt \mapsto nbh\}\}$

a6 : $updateNbrs := updateNbrs \cup \{f \mapsto nb\}$

end

Event *receive_dup_controlPkt* $\widehat{=}$

//(ctl) each neighbour node including sink receives a duplicated control packet from channel

extends *receive_dup_controlPkt*

any

nb

pkt

f

where

g1 : $pkt \in dom(ctlNeighbours)$

g2 : $pkt \mapsto nb \in ctlNeighbours$

g3 : $nb \in dom(floodTbl) \wedge pkt \in floodTbl(nb)$

g9 : $type(pkt) \in CONTROL$

g11 : $pkt \in dom(pktFwdr) \wedge f = pktFwdr(pkt)$

g12 : $f \in ND \wedge nb \in ND$

g13 : $f \mapsto nb \notin updateNbrs$

then

a1 : $ctlNeighbours := ctlNeighbours \setminus \{pkt \mapsto nb\}$

a4 : $updateNbrs := updateNbrs \cup \{f \mapsto nb\}$

end

Event *sink_recv_dataPkt* $\widehat{=}$

//(ctl) a sink receives a data packet from a channel

extends *sink_recv_dataPkt*

any

s

pkt

nb

where

g1 : $s = initialSrcAddr(pkt)$

g2 : $s \neq Sink$

g3 : $pkt \in ran(sentDown \cup sentUp) \setminus sinkBuff$

g4 : $nb = Sink$

g5 : $pkt \mapsto nb \in ctlNeighbours$

g6 : $nb \in dom(floodTbl) \wedge pkt \notin floodTbl(nb)$

g7 : $type(pkt) = DATA$

then

a1 : $sinkBuff := sinkBuff \cup \{pkt\}$

a2 : $ctlNeighbours := ctlNeighbours \setminus \{pkt \mapsto nb\}$

a3 : $floodTbl(nb) := floodTbl(nb) \cup \{pkt\}$

end

Event *sink_recv_controlPkt* $\widehat{=}$

//(ctl) a sink receives a control packet from channel

extends *sink_recv_controlPkt*

any

s

pkt

nb

f

where

g1 : $s = initialSrcAddr(pkt)$

g2 : $s \neq Sink$

g3 : $pkt \in ran(sentDown \cup sentUp) \setminus sinkBuff$

g4 : $nb = Sink$

g5 : $pkt \mapsto nb \in ctlNeighbours$

g6 : $nb \in dom(floodTbl) \wedge pkt \notin floodTbl(nb)$

g?7 : $type(pkt) \neq DATA$

g8 : $pkt \in dom(pktFwdr) \wedge f = pktFwdr(pkt)$

g9 : $f \in ND \wedge nb \in ND$

g10 : $f \mapsto nb \notin updateNbrs$

then

a1 : $sinkBuff := sinkBuff \cup \{pkt\}$

a2 : $ctlNeighbours := ctlNeighbours \setminus \{pkt \mapsto nb\}$

a3 : $floodTbl(nb) := floodTbl(nb) \cup \{pkt\}$

a4 : $updateNbrs := updateNbrs \cup \{f \mapsto nb\}$

end

Event *add_newEntry* $\widehat{=}$

//(ctl) adding new entry into the neighbour table

extends *add_newEntry*

any

x

y

where

g1 : $updateNbrs \neq \varnothing$

g2 : $x \in ND \wedge y \in ND \wedge x \mapsto y \in updateNbrs$

g4 : $y \mapsto x \notin neighbourTbl$

g5 : $y \mapsto x \mapsto 0 \notin missed$

g6 : $y \mapsto x \mapsto 0 \notin received$

g7 : $y \mapsto x \mapsto 0 \notin lastSeqno$

g8 : $y \mapsto x \mapsto 0 \notin receiveEst$

g9 : $y \mapsto x \mapsto 0 \notin sentEst$

g10 : $y \mapsto x \mapsto LIVELINESS \notin liveliness$

then

a1 : $neighbourTbl := neighbourTbl \cup \{y \mapsto x\}$

a2 : $missed := missed \cup \{y \mapsto x \mapsto 0\}$

a3 : $received := received \cup \{y \mapsto x \mapsto 0\}$

a4 : $lastSeqno := lastSeqno \cup \{y \mapsto x \mapsto 0\}$

a5 : $receiveEst := receiveEst \cup \{y \mapsto x \mapsto 0\}$

a6 : $sentEst := sentEst \cup \{y \mapsto x \mapsto 0\}$

//set sentEst of each nighbour node to be zero

a7 : $liveliness := liveliness \cup \{y \mapsto x \mapsto LIVELINESS\}$

//set liveliness of each nighbour node to be zero

end

Event *update_nbr* $\widehat{=}$

//(ctl) update receiving information into the neighbour table

extends *update_nbr*

any

x

y

sNo

$delta$

pkt

where

g1 : $updateNbrs \neq \varnothing$

g2 : $x \in ND \wedge y \in ND \wedge x \mapsto y \in updateNbrs$

g3 : $y \mapsto x \in neighbourTbl$

g4 : $pkt \mapsto x \in pktFwdr$

g5 : $sNo = netSeqNo(pkt)$

g6 : $delta = sNo - lastSeqno(y \mapsto x) - 1$

g7 : $delta \geq 0$

g8 : $type(pkt) = BEACON$

g9 : $pkt \in ran(sentUp)$

then

a1 : $received := received \lhd\!\!\!+ \{y \mapsto x \mapsto (received(y \mapsto x) + 1)\}$

a2 : $lastSeqno := lastSeqno \lhd\!\!\!+ \{y \mapsto x \mapsto sNo\}$

a3 : $missed := missed \lhd\!\!\!+ \{y \mapsto x \mapsto delta\}$

a4 : $updateNbrs := updateNbrs \setminus \{x \mapsto y\}$

end

Event *update_route* $\widehat{=}$

//(ctl) updating information from route packet into the neighbour table

extends *update_nbr2*

any

x

y

pkt

where

g1 : $updateNbrs \neq \varnothing$

g2 : $x \in ND \wedge y \in ND \wedge x \mapsto y \in updateNbrs$

g3 : $y \mapsto x \in neighbourTbl$

g4 : $pkt \mapsto x \in pktFwdr$

g5 : $type(pkt) \neq DATA \wedge type(pkt) \neq BEACON$

g6 : $pkt \in ran(sentUp)$

g7 : $y \mapsto x \in dom(sentEst)$

g8 : $x \mapsto y \in dom(receiveEst)$

g9 : $y \mapsto x \in dom(liveliness)$

then

a1 : $updateNbrs := updateNbrs \setminus \{x \mapsto y\}$

a2 : $sentEst(y \mapsto x) := receiveEst(x \mapsto y)$

//updating the transmission quality ratio with the reception quality ratio attached in route update packets

a3 : $liveliness(y \mapsto x) := LIVELINESS$

//set liveliness field back to liveliness threshold

end

Event *decrease_liveliness* $\widehat{=}$

//(ctl) decreasing liveliness of each neighbour in neighbour table

any

x

y

$live$

where

g1 : $estNbrs = neighbourTbl$

g2 : $x \mapsto y \in checkedLiveNbrs$

g3 : $x \mapsto y \in dom(liveliness)$

g4 : $live = liveliness(x \mapsto y) \wedge live > 0$

then

a1 : $liveliness := liveliness \Leftarrow \{x \mapsto y \mapsto (live - 1)\}$

a2 : $checkedLiveNbrs := checkedLiveNbrs \setminus \{x \mapsto y\}$

end

Event *exclude_deadNbr* $\widehat{=}$

//(ctl) excluding dead neighbour from neighbour table

any

x

y

$live$

where

g1 : $estNbrs = neighbourTbl$

g2 : $x \mapsto y \in checkedLiveNbrs$
g3 : $x \mapsto y \in dom(liveliness)$
g4 : $live = liveliness(x \mapsto y) \wedge live = 0$
g5 : $y \mapsto x \in dom(sentEst)$

then

a1 : $sentEst(y \mapsto x) := 1$
a2 : $checkedLiveNbrs := checkedLiveNbrs \setminus \{x \mapsto y\}$

end

Event *est_due* $\widehat{=}$

//(ctl) link quality estimation timer due

extends *est_due*

any

x

where

g1 : $x \in ND \setminus estNDs$
g2 : $x \in dom(totalSentBcon) \wedge totalSentBcon(x) > 0$
g3 : $totalSentBcon(x) mod EST\_RATIO = 0$

then

a1 : $estNDs := estNDs \cup \{x\}$

end

Event *update_est* $\widehat{=}$

//(ctl) calculating and updating reception ratio in case no lost packets or lost packets in the middle of transmission

extends *update_est*

any

nd
nb
r
m
rt
$newAve$

where

g1 : $nd \in estNDs$
g2 : $nb \in ran(\{nd\} \lhd neighbourTbl)$
g3 : $nd \in dom(totalSentBcon) \wedge totalSentBcon(nd) > 0$
g4 : $nd \mapsto nb \in dom(lastSeqno) \wedge lastSeqno(nd \mapsto nb) > 0$
g5 : $r = received(nd \mapsto nb)$
g6 : $m = missed(nd \mapsto nb)$
g8 : $rt = EST\_RATIO$
g9 : $newAve = ((r * 100)/rt)$
g10 : $nd \mapsto nb \in dom(receiveEst) \wedge newAve > 0$

g11 : $nd \mapsto nb \in dom(missed)$
g12 : $nd \mapsto nb \in dom(received)$
g13 : $nd \mapsto nb \notin estNbrs$
g14 : $estNbrs \neq neighbourTbl$
g15 : $nd \mapsto nb \notin checkedLiveNbrs$

then

a1 : $receiveEst := receiveEst \lhd\!\!\!- \{nd \mapsto nb \mapsto newAve\}$
a2 : $missed := missed \lhd\!\!\!- \{nd \mapsto nb \mapsto 0\}$
a3 : $received := received \lhd\!\!\!- \{nd \mapsto nb \mapsto 0\}$
a4 : $estNbrs := estNbrs \cup \{nd \mapsto nb\}$
a7 : $checkedLiveNbrs := checkedLiveNbrs \cup \{nd \mapsto nb\}$

end

Event *update_est_nothing* $\widehat{=}$

//(ctl) for case the dead neighbour node affecting received become zero

extends *update_est_nothing*

any

nd
nb
r
m
rt
$newAve$

where

g1 : $nd \in estNDs$
g2 : $nb \in ran(\{nd\} \lhd neighbourTbl)$
g3 : $nd \in dom(totalSentBcon) \wedge totalSentBcon(nd) > 0$
g4 : $nd \mapsto nb \in dom(lastSeqno) \wedge lastSeqno(nd \mapsto nb) > 0$
g5 : $r = received(nd \mapsto nb)$
g6 : $m = missed(nd \mapsto nb)$
g7 : $rt = EST\_RATIO \wedge rt > 0$
g8 : $newAve = ((r * 100)/rt)$
g9 : $newAve = 0$
g10 : $nd \mapsto nb \notin estNbrs$
g11 : $estNbrs \neq neighbourTbl$
g12 : $nd \mapsto nb \notin checkedLiveNbrs$

then

a1 : $estNbrs := estNbrs \cup \{nd \mapsto nb\}$
a2 : $checkedLiveNbrs := checkedLiveNbrs \cup \{nd \mapsto nb\}$

end

Event *finish_tx_pkt* $\widehat{=}$

//(ctl) finishing packet transmission

extends *finish_tx_pkt*

any

f

pkt

where

g1 : $\{pkt\} \lhd ctlNeighbours = \varnothing$

g2 : $pkt \in ran(sentUp) \wedge pkt \notin ran(sentDown) \wedge pkt \in ran(ndBuff)$

g3 : $f \in dom(sentUp) \wedge f \mapsto pkt \in sentUp$

g4 : $type(pkt) = DATA \vee type(pkt) = BEACON$

then

a1 : $sentUp := sentUp \setminus \{f \mapsto pkt\}$

end

Event *final_tx_pkt* $\widehat{=}$

//(ctl) the transmitted packet is removed from middleare indicating completiion of transmission mechanism

extends *final_tx_pkt*

any

pkt

where

g1 : $\{pkt\} \lhd ctlNeighbours = \varnothing$

g2 : $pkt \in ran(sentUp) \wedge pkt \notin ran(sentDown) \wedge pkt \notin ran(ndBuff)$

g3 : $type(pkt) = DATA \vee type(pkt) = BEACON$

then

a1 : $sentUp := sentUp \rhd \{pkt\}$

end

Event *lose_all_neighbours* $\widehat{=}$

//(env) channel loses all neighbours

extends *lose_all_neighbours*

any

s

nbs

f

pkt

where

g1 : $channel \neq \varnothing$

g2 : $pkt \in ran(channel)$

g3 : $s = initialSrcAddr(pkt)$

g4 : $f \mapsto pkt \in channel$

g5 : $nbs = wsnLinks[\{f\}] \wedge nbs = \varnothing$

then

a1 : $envNeighbours := envNeighbours \cup \{pkt \mapsto FAILED\_XMIT\}$

end

Event *lose_pkt* $\widehat{=}$

//(ctl) each node records a lost packet

extends *lose_pkt*

any

f

pkt

where

g1 : $pkt \in ran(sentUp) \wedge pkt \notin ran(sentDown) \wedge pkt \notin ran(ndBuff)$

g2 : $pkt \notin ran(recLostPkts)$

g3 : $f \mapsto pkt \in sentUp \wedge f \mapsto pkt \notin sentDown$

g4 : $f \mapsto pkt \notin recLostPkts$

g5 : $ran(\{pkt\} \lhd ctlNeighbours) = \{FAILED\_XMIT\}$

then

a1 : $sentUp := sentUp \rhd \{pkt\}$

a2 : $recLostPkts := recLostPkts \cup \{f \mapsto pkt\}$

end

END

A.6 Fifth refinement: parent selection and unicast packet forwarding mechanism

A.6.1 Context:C4

CONTEXT C4

EXTENDS C3

CONSTANTS

$QThreshold$

$Qlow$

$Qmed$

$Qhigh$

$INFINITY$

AXIOMS

axm2_1 : $QThreshold \subseteq \mathbb{N}$

axm2_2 : $Qlow \in \mathbb{N}$

axm2_3 : $Qmed \in \mathbb{N}$

axm2_4 : $Qhigh \in \mathbb{N}$

axm2_5 : $QThreshold = \{Qlow, Qmed, Qhigh\}$
//link quality thresholds: lower, medium and higher
axm2_6 : $MAX\_DIST \in \mathbb{N}$ //maximal path cost value
axm2_7 : $MAX\_DIST = 9999$

END

A.6.2 Context:TCL

CONTEXT TCL

/* Defines objects, root object and
transitive closure of relations on objects,
and introduces some theorems used for discharging POs. */

EXTENDS C2

CONSTANTS

tcl transitive closure of an objrel

AXIOMS

axm_tcl_1 : $tcl \in WSN \rightarrow WSN$
axm_tcl_2 : $\forall r \cdot (r \in WSN \Rightarrow r \subseteq tcl(r))$ //r is included in tcl(r)
axm_tcl_3 : $\forall r \cdot (r \in WSN \Rightarrow r; tcl(r) \subseteq tcl(r))$
//forward composition of tcl(r) with r is included in tcl(r)
axm_tcl_4 : $\forall r, t \cdot (r \in WSN \wedge r \subseteq t \wedge r; t \subseteq t \Rightarrow tcl(r) \subseteq t)$
//tcl(r) is the smallest relation
axm_tcl_5 : $tcl(\varnothing) = \varnothing$

END

A.6.3 Machine:M5

MACHINE M5

REFINES M4

SEES C4, TCL

VARIABLES

$sinkBuff$ //(ctl) a sink buffer
$ndBuff$ //(ctl) nodes' waiting buffer
$sensingNDs$ //(ctl) sensing nodes
$sensedPkts$ //(ctl) sensed packets
$senseBuff$ //(ctl) sense buffer
$floodedPkts$ //(ctl) flooded packets

$floodTbl$ //(ctl)historical received packets used for detect duplicated packets
$ctlNeighbours$ //(ctl) neighbour lists
$recLostPkts$ //(ctl) recorded lost packet at each node
$envSensedFlg$ //(env) sensing period
$envData$ //(env) environmental/conditional data
$ctlSensedFlg$ //(ctl) sensing period
$sentDown$ //(ctl) transmitted packets down to channel
$sentUp$ //(ctl) transmitted packets up from channel
$channel$ //(env) channel
$envNeighbours$ //(env) neighbour list
$wsnLinks$ //(env) dynamic WSN links
$floodFlg$ //(ctl) flooded flag
$dataSeqNo$ //(ctl) local nodes' current data sequence number at application layer
$floodSeqNo$ //(ctl) local nodes' current broadcast sequence number at application layer
$nbHops$ //(ctl) neighbour's hop count
$crashedLinks$ //(env) crashed WSN Links
$pktSrc$ //(ctl) src address attached in transmitted packet
$pktSeqNo$ //(ctl) sequence number attached in transmitted packet
$pktFwdr$ //(ctl) fowarder attached in transmitted packet
$pktData$ //(ctl) data attached in transmitted packet
$pktNbHops$ //(ctl) hop count attached in transmitted packet
$vPktSeqNo$ //(env) virtual packet at channel
$vPktSrc$ //(env) virtual packet at channel
$vPktFwdr$ //(env) virtual packet at channel
$vPktData$ //(env) virtual packet at channel
$vPktNbHops$ //(env) virtual packet at channel
$totalSentBcon$ //(ctl) number of sent control packets, used for cost-estimation
$netSeqNo$ //(ctl) link sequence no in a packet, used for cost-estimation
$linkSeqNo$ //(ctl) link sequence number of each node
$updateNbrs$ //(ctl) for updating neighbours in the neighbour table
$neighbourTbl$ //(ctl)neighbour table structure
$missed$ //(ctl) neighbour table: field missed (number of lost packets)
$received$ //(ctl) neighbour table: field received (number of received packets)
$lastSeqno$ //(ctl) neighbour table: field lastSeqno in neighbour table
$receiveEst$ //(ctl) neighbour talble: field receiveEst (reception link quality)
$estNDs$ //(ctl) estimated nodes
$estNbrs$ //(ctl) estimated neighbours

$bcastRouTimer$ //(ctl) broadcast route timer

$bcastRouNodes$ //(ctl) route broadcast nodes

$routeSeqNo$ //(ctl) sequence number for route packet at each node

$sentEst$ //(ctl) neighbour table: field sentEst (transmission quality) in neighbour table

$liveliness$ //(ctl) neighbour table: field liveliness in neighbour table

$checkedLiveNbrs$ //(ctl) checked liveliness nodes

$chooseParentTimer$ //(ctl) choose parent timer

$nodes$ //(ctl) nodes in the route tree

$completedRoute$ //(ctl) completed route flag

$cpCost$ //(ctl) current parent cost

$cRouteTree$ //(ctl) the current route tree

$calPCostNDs$ //(ctl) for calculated cost nodes

$calPCost$ //(ctl) calculated cost for finding min

$pktDestAddr$ //(ctl) packet field destination address

$vPktDestAddr$ //(env) packet field destination addr for channel

$calPCostFlg$ //(ctl) calculateded path cost

$parent$ //(ctl) neighbour table: parent of neighbour

$cost$ //(ctl) neithbour table: path cost

$deadNbrs$ //(ctl) dead neighbours

$chosenNDs$ //(ctl) chosen nodes for identifying a parent

$Nbrs$ //(ctl) list of neighbours

INVARIANTS

inv5_1 : $chooseParentTimer \in BOOL$

inv5_21 : $nodes \subseteq ND$

inv5_22 : $cRouteTree \in (nodes \setminus \{Sink\}) \rightarrow ND$

inv5_3 : $Nbrs \subseteq ND$

inv5_4 : $completedRoute \in BOOL$

inv5_6 : $cpCost \in ND \rightarrow (\mathbb{N} \cup \{MAX\_DIST\})$

inv5_7 : $calPCostNDs \in ND \leftrightarrow ND$

inv5_8 : $calPCost \in calPCostNDs \rightarrow \mathbb{N}$

inv5_9 : $vPktDestAddr \in PKT \nrightarrow (ND \cup \{BROADCAST\})$

inv5_10 : $calPCostFlg \in BOOL$

inv5_11 : $pktDestAddr \in PKT \nrightarrow (ND \cup \{BROADCAST\})$

inv5_12 : $parent \in neighbourTbl \rightarrow (ND \cup \{BROADCAST\})$

inv5_13 : $cost \in neighbourTbl \rightarrow \mathbb{N}$

inv5_14 : $deadNbrs \subseteq ND$

saf5_102 : $\forall S \cdot S \subseteq cRouteTree^{-1}[S] \Rightarrow S = \varnothing$

saf5_103 : $completedRoute = TRUE \Rightarrow nodes = ND$

mth1 : $\forall T \cdot (Sink \in T \wedge cRouteTree^{-1}[T] \subseteq T \Rightarrow nodes \subseteq T)$
//help to prove @mth3

mth2 : $nodes \subseteq \{Sink\} \cup (tcl(cRouteTree))^{-1}[\{Sink\}]$
//help to prove @mth3

mth3 : $nodes \setminus \{Sink\} \subseteq (tcl(cRouteTree))^{-1}[\{Sink\}]$
//reachability property

EVENTS

Initialisation

extended

begin

int0_3 : $sinkBuff := \varnothing$

int1_1 : $ndBuff := \varnothing$

int1_3 : $sensingNDs := \varnothing$

int1_4 : $sensedPkts := \varnothing$

int1_5 : $senseBuff := (ND \setminus \{Sink\}) \times \{\varnothing\}$

int1_6 : $floodedPkts := \varnothing$

int1_7 : $floodTbl := ND \times \{\varnothing\}$

int1_8 : $ctlNeighbours := \varnothing$

int1_9 : $envSensedFlg := (ND \setminus \{Sink\}) \times \{FALSE\}$

int1_10 : $envData := (ND \setminus \{Sink\}) \times \{0\}$

int1_11 : $ctlSensedFlg := (ND \setminus \{Sink\}) \times \{FALSE\}$

int1_12 : $floodFlg := ND \times \{FALSE\}$

inv1_20 : $recLostPkts := \varnothing$

int2_1 : $sentUp := \varnothing$

int2_2 : $sentDown := \varnothing$

int2_3 : $channel := \varnothing$

int2_10 : $envNeighbours := \varnothing$

int2_11 : $wsnLinks := \varnothing$

int2_12 : $crashedLinks := \varnothing$

int2_13 : $dataSeqNo := ND \times \{0\}$

int2_14 : $floodSeqNo := ND \times \{0\}$

int2_15 : $nbHops := \varnothing$

int2_16 : $pktSeqNo := \varnothing$

int2_17 : $pktSrc := \varnothing$

int2_18 : $pktFwdr := \varnothing$

int2_19 : $pktData := \varnothing$

int2_20 : $pktNbHops := \varnothing$

int2_21 : $vPktSeqNo := \varnothing$

int2_22 : $vPktSrc := \varnothing$

int2_23 : $vPktFwdr := \varnothing$
int2_24 : $vPktData := \varnothing$
int2_25 : $vPktNbHops := \varnothing$
int3_1 : $totalSentBcon := ND \times \{0\}$
int3_2 : $netSeqNo := PKT \times \{0\}$
int3_3 : $updateNbrs := \varnothing$
int3_4 : $neighbourTbl := \varnothing$
int3_5 : $missed := \varnothing$
int3_6 : $received := \varnothing$
int3_7 : $lastSeqno := \varnothing$
int3_8 : $receiveEst := \varnothing$
int3_9 : $linkSeqNo := ND \times \{0\}$
int3_11 : $estNDs := \varnothing$
int3_12 : $estNbrs := \varnothing$
int4_1 : $sentEst := \varnothing$
int4_2 : $bcastRouTimer := FALSE$
int4_3 : $bcastRouNodes := \varnothing$
int4_4 : $routeSeqNo := ND \times \{0\}$
int4_5 : $liveliness := \varnothing$
int4_6 : $checkedLiveNbrs := \varnothing$
int5_1 : $chooseParentTimer := FALSE$
int5_21 : $nodes := \{Sink\}$
int5_22 : $cRouteTree := \varnothing$
int5_3 : $Nbrs := \varnothing$
int5_4 : $completedRoute := FALSE$
int5_6 : $cpCost := ((ND \setminus \{Sink\}) \times \{MAX\_DIST\}) \cup \{Sink \mapsto 0\}$
int5_8 : $calPCostNDs := \varnothing$
int5_9 : $calPCost := \varnothing$
int5_11 : $pktDestAddr := \varnothing$
int5_12 : $vPktDestAddr := \varnothing$
int5_16 : $calPCostFlg := FALSE$
int5_19 : $deadNbrs := \varnothing$
int5_23 : $parent := \varnothing$
int5_24 : $cost := \varnothing$
int5_26 : $chosenNDs := \varnothing$

end

Event *add_link* $\widehat{=}$

//(env) channel sets a topology

extends *add_link*

any

x

y

where

g1 : $x \in ND \wedge y \in ND$

g2 : $x \mapsto y \notin wsnLinks$

g3 : $x \neq y$

then

a1 : $wsnLinks := wsnLinks \cup \{x \mapsto y\}$

end

Event *disconnect_link* $\widehat{=}$

//(env) disconnecting a link

extends *disconnect_link*

any

x

y

where

g1 : $x \in ND \wedge y \in ND$

g2 : $x \mapsto y \in wsnLinks$

g3 : $x \mapsto y \notin crashedLinks$

then

a1 : $crashedLinks := crashedLinks \cup \{x \mapsto y\}$

a2 : $wsnLinks := wsnLinks \setminus \{x \mapsto y\}$

end

Event *recover_link* $\widehat{=}$

//(env) recovering a link

extends *recover_link*

any

x

y

where

g1 : $x \in ND \wedge y \in ND$

g2 : $x \mapsto y \in crashedLinks$

g3 : $x \mapsto y \notin wsnLinks$

g4 : $x \neq y$

then

a1 : $wsnLinks := wsnLinks \cup \{x \mapsto y\}$

a2 : $crashedLinks := crashedLinks \setminus \{x \mapsto y\}$

end

Event *start_sensing* $\widehat{=}$

//(env) activating each node to sense data

extends *start_sensing*

any

x

sd

where

g1 : $x \in ND \setminus \{Sink\}$

g2 : $envSensedFlg(x) = FALSE$

g3 : $sd \in randomFn$

then

a1 : $envSensedFlg(x) := TRUE$

a2 : $envData(x) := sd$

end

Event *sensing* $\widehat{=}$

//(shd) each node senses a data from environment

extends *sensing*

any

x

sf

sd

where

g1 : $x \in ND \setminus \{Sink\}$

g2 : $sf = envSensedFlg(x)$

g3 : $envSensedFlg(x) = TRUE$

g4 : $ctlSensedFlg(x) = FALSE$

g5 : $sd = envData(x)$

then

a1 : $ctlSensedFlg(x) := sf$

a2 : $senseBuff(x) := senseBuff(x) \cup \{sd\}$

end

Event *finish_sensing* $\widehat{=}$

//(shd) finishing sensing data process

extends *finish_sensing*

any

x

where

g1 : $x \in dom(envSensedFlg)$

g2 : $envSensedFlg(x) = TRUE$

g3 : $ndBuff = \varnothing \wedge (sentDown \cup sentUp) = \varnothing$

g4 : $ctlSensedFlg(x) = TRUE$

g5 : $sensingNDs = \{x\}$

then

a1 : $ctlSensedFlg(x) := FALSE$
a2 : $envSensedFlg(x) := FALSE$
a3 : $sensingNDs := \varnothing$

end

Event *create_dataPkt* $\widehat{=}$

//(ctl) creating a data packet

extends *create_dataPkt*

any

s
pkt
sd
sno
nbh

where

g1 : $pkt \in PKT \wedge s \in ND \setminus \{Sink\}$
g2 : $pkt \notin ran(ndBuff \cup (sentUp \cup sentDown))$
g3 : $s \notin sensingNDs \wedge pkt \notin sensedPkts \cup floodedPkts$
g4 : $s = initialSrcAddr(pkt)$
g5 : $s \in dom(floodTbl) \wedge pkt \notin floodTbl(s)$
g6 : $type(pkt) = DATA$
g7 : $ctlSensedFlg(s) = TRUE$
g8 : $senseBuff \neq \varnothing \wedge sd \in senseBuff(s)$
g9 : $sno = dataSeqNo(s) + 1 \wedge s \in dom(dataSeqNo)$
g10 : $nbh \in \mathbb{Z} \wedge nbh = -1$
g11 : $s \mapsto \{pkt \mapsto nbh\} \notin nbHops$
g12 : $pkt \notin dom(pktSeqNo)$
g13 : $pkt \notin dom(pktSrc)$
g14 : $pkt \notin dom(pktFwdr)$
g15 : $pkt \notin dom(pktData)$
g16 : $pkt \notin dom(pktNbHops)$
g17 : $pkt \notin dom(pktDestAddr)$

then

a1 : $sensingNDs := sensingNDs \cup \{s\}$
a2 : $sensedPkts := sensedPkts \cup \{pkt\}$
a3 : $ndBuff := ndBuff \cup \{s \mapsto pkt\}$
a4 : $floodTbl(s) := floodTbl(s) \cup \{pkt\}$
a5 : $dataSeqNo := dataSeqNo \lhd\!\!\!- \{s \mapsto sno\}$
a6 : $nbHops := nbHops \cup \{s \mapsto \{pkt \mapsto nbh\}\}$
a7 : $senseBuff(s) := senseBuff(s) \setminus \{sd\}$
a8 : $pktSeqNo := pktSeqNo \lhd\!\!\!- \{pkt \mapsto sno\}$
a9 : $pktSrc := pktSrc \cup \{pkt \mapsto s\}$

a10 : $pktFwdr := pktFwdr \cup \{pkt \mapsto s\}$
a11 : $pktData := pktData \cup \{pkt \mapsto sd\}$
a12 : $pktNbHops := pktNbHops \cup \{pkt \mapsto nbh\}$
a13 : $pktDestAddr := pktDestAddr \cup \{pkt \mapsto BROADCAST\}$

end

Event *create_bconPkt* $\widehat{=}$

//create a beacon packet

extends *create_bconPkt*

any

s

pkt

sno

nbh

where

g1 : $pkt \in PKT$
g2 : $pkt \notin ran(ndBuff \cup (sentUp \cup sentDown))$
g3 : $pkt \notin (sensedPkts \cup floodedPkts)$
g4 : $s = initialSrcAddr(pkt)$
g5 : $s \in dom(floodTbl) \wedge pkt \notin floodTbl(s)$
g6 : $type(pkt) \in CONTROL$
g7 : $floodFlg(s) = TRUE$
g8 : $sno = floodSeqNo(s) + 1 \wedge s \in dom(floodSeqNo)$
g9 : $nbh \in \mathbb{Z} \wedge nbh = -1$
g10 : $s \mapsto \{pkt \mapsto nbh\} \notin nbHops$
g11 : $pkt \notin dom(pktSeqNo)$
g12 : $pkt \notin dom(pktSrc)$
g13 : $pkt \notin dom(pktFwdr)$
g14 : $pkt \notin dom(pktData)$
g15 : $pkt \notin dom(pktNbHops)$
g16 : $s = Sink$
g17 : $type(pkt) = BEACON$
g18 : $pkt \notin dom(pktDestAddr)$

then

a1 : $floodedPkts := floodedPkts \cup \{pkt\}$
a2 : $ndBuff := ndBuff \cup \{s \mapsto pkt\}$
a3 : $floodTbl(s) := floodTbl(s) \cup \{pkt\}$
a4 : $floodSeqNo := floodSeqNo \Leftarrow \{s \mapsto sno\}$
a5 : $nbHops := nbHops \cup \{s \mapsto \{pkt \mapsto nbh\}\}$
a6 : $pktSeqNo := pktSeqNo \Leftarrow \{pkt \mapsto sno\}$
a7 : $pktFwdr := pktFwdr \cup \{pkt \mapsto s\}$
a8 : $pktSrc := pktSrc \cup \{pkt \mapsto s\}$

a9 : $pktData := pktData \cup \{pkt \mapsto 0\}$

a10 : $pktNbHops := pktNbHops \cup \{pkt \mapsto nbh\}$

a11 : $pktDestAddr := pktDestAddr \cup \{pkt \mapsto BROADCAST\}$

end

Event *bcastRou_due* $\widehat{=}$

//broadcast route timer due

extends *bcastRou_due*

when

g1 : $estNDs = ND$

g2 : $totalSentBcon \neq \varnothing$

g3 : $estNbrs = neighbourTbl$

g4 : $bcastRouTimer = FALSE$

g5 : $checkedLiveNbrs = \varnothing$

then

a1 : $estNDs := \varnothing$

a3 : $estNbrs := \varnothing$

a4 : $totalSentBcon := ND \times \{0\}$

a5 : $bcastRouTimer := TRUE$

end

Event *reset_bcastRouTimer* $\widehat{=}$

//(ctl) reseting broadcast route timer for the next round

extends *reset_bcastRouTimer*

when

g1 : $bcastRouTimer = TRUE$

g2 : $bcastRouNodes = ND$

then

a1 : $bcastRouTimer := FALSE$

a2 : $bcastRouNodes := \varnothing$

end

Event *create_routePkt* $\widehat{=}$

//(ctl) create a route packet

extends *create_routePkt*

any

s

pkt

sno

nbh

where

g1 : $pkt \in PKT$

g2 : $pkt \notin ran(ndBuff \cup (sentUp \cup sentDown))$

g3 : $pkt \notin (sensedPkts \cup floodedPkts)$
g4 : $s = initialSrcAddr(pkt)$
g5 : $s \in dom(floodTbl) \wedge pkt \notin floodTbl(s)$
g6 : $type(pkt) \in CONTROL$
g7 : $floodFlg(s) = TRUE$
g8 : $sno = floodSeqNo(s) + 1 \wedge s \in dom(floodSeqNo)$
g9 : $nbh \in \mathbb{Z} \wedge nbh = -1$
g10 : $s \mapsto \{pkt \mapsto nbh\} \notin nbHops$
g11 : $pkt \notin dom(pktSeqNo)$
g12 : $pkt \notin dom(pktSrc)$
g13 : $pkt \notin dom(pktFwdr)$
g14 : $pkt \notin dom(pktData)$
g15 : $pkt \notin dom(pktNbHops)$
g16 : $type(pkt) = ROUTE$
g17 : $bcastRouTimer = TRUE$
g18 : $s \notin bcastRouNodes$
g19 : $pkt \notin dom(pktDestAddr)$

then

a1 : $floodedPkts := floodedPkts \cup \{pkt\}$
a2 : $ndBuff := ndBuff \cup \{s \mapsto pkt\}$
a3 : $floodTbl(s) := floodTbl(s) \cup \{pkt\}$
a4 : $floodSeqNo := floodSeqNo \Leftarrow \{s \mapsto sno\}$
a5 : $nbHops := nbHops \cup \{s \mapsto \{pkt \mapsto nbh\}\}$
a6 : $pktSeqNo := pktSeqNo \Leftarrow \{pkt \mapsto sno\}$
a7 : $pktFwdr := pktFwdr \cup \{pkt \mapsto s\}$
a8 : $pktSrc := pktSrc \cup \{pkt \mapsto s\}$
a9 : $pktData := pktData \cup \{pkt \mapsto 0\}$
a10 : $pktNbHops := pktNbHops \cup \{pkt \mapsto nbh\}$
a11 : $bcastRouNodes := bcastRouNodes \cup \{s\}$
a12 : $pktDestAddr := pktDestAddr \cup \{pkt \mapsto BROADCAST\}$

end

Event *start_tx_dataPkt_bct* $\widehat{=}$

//(ctl) starting broadcasting a data packet to a channel

extends *start_tx_dataPkt*

any

x
pkt
nbh

where

g1 : $x \mapsto pkt \in ndBuff$
g2 : $x \mapsto pkt \notin sentDown$

g3 : $x \mapsto pkt \notin sentUp$
g4 : $pkt \in dom(pktFwdr)$
g5 : $x \mapsto \{pkt \mapsto nbh\} \in nbHops$
g6 : $pkt \in dom(pktNbHops)$
g7 : $type(pkt) = DATA$
g8 : $completedRoute = FALSE$

then

a1 : $ndBuff := ndBuff \setminus \{x \mapsto pkt\}$
a2 : $sentDown := sentDown \cup \{x \mapsto pkt\}$
a3 : $pktFwdr := pktFwdr \lhd\!\!\!- \{pkt \mapsto x\}$
a4 : $pktNbHops := pktNbHops \lhd\!\!\!- \{pkt \mapsto nbh\}$
a5 : $nbHops := nbHops \setminus \{x \mapsto \{pkt \mapsto nbh\}\}$

end

Event *start_tx_dataPkt_fwd* $\widehat{=}$

//(ctl) starting unicasting a data packet to a channel via route

extends *start_tx_dataPkt*

any

x
pkt
nbh
y

where

g1 : $x \mapsto pkt \in ndBuff$
g2 : $x \mapsto pkt \notin sentDown$
g3 : $x \mapsto pkt \notin sentUp$
g4 : $pkt \in dom(pktFwdr)$
g5 : $x \mapsto \{pkt \mapsto nbh\} \in nbHops$
g6 : $pkt \in dom(pktNbHops)$
g7 : $type(pkt) = DATA$
g8 : $completedRoute = TRUE$
g9 : $x \in dom(cRouteTree)$
g10 : $y = cRouteTree(x)$ //retrieving next node y of node x
g11 : $pkt \in dom(pktDestAddr)$

then

a1 : $ndBuff := ndBuff \setminus \{x \mapsto pkt\}$
a2 : $sentDown := sentDown \cup \{x \mapsto pkt\}$
a3 : $pktFwdr := pktFwdr \lhd\!\!\!- \{pkt \mapsto x\}$
a4 : $pktNbHops := pktNbHops \lhd\!\!\!- \{pkt \mapsto nbh\}$
a5 : $nbHops := nbHops \setminus \{x \mapsto \{pkt \mapsto nbh\}\}$
a6 : $pktDestAddr := pktDestAddr \lhd\!\!\!- \{pkt \mapsto y\}$ //assigning next node y in a packet

end

Event *start_tx_bconPkt* $\widehat{=}$

//(ctl) starting transmiting a beacon packet to a channel

extends *start_tx_bconPkt*

any

x

pkt

nbh

$lsno$

where

g1 : $x \mapsto pkt \in ndBuff$

g2 : $x \mapsto pkt \notin sentDown$

g3 : $x \mapsto pkt \notin sentUp$

g4 : $pkt \in dom(pktFwdr)$

g5 : $x \mapsto \{pkt \mapsto nbh\} \in nbHops$

g6 : $pkt \in dom(pktNbHops)$

g7 : $type(pkt) = BEACON$

g8 : $x \in dom(totalSentBcon) \wedge totalSentBcon(x) \geq 0$

g9 : $lsno = linkSeqNo(x) + 1 \wedge x \in dom(linkSeqNo)$

then

a1 : $ndBuff := ndBuff \setminus \{x \mapsto pkt\}$

a2 : $sentDown := sentDown \cup \{x \mapsto pkt\}$

a3 : $pktFwdr := pktFwdr \lhd\!\!\!- \{pkt \mapsto x\}$

a4 : $pktNbHops := pktNbHops \lhd\!\!\!- \{pkt \mapsto nbh\}$

a5 : $nbHops := nbHops \setminus \{x \mapsto \{pkt \mapsto nbh\}\}$

a6 : $totalSentBcon(x) := totalSentBcon(x) + 1$

a7 : $linkSeqNo := linkSeqNo \lhd\!\!\!- \{x \mapsto lsno\}$

a8 : $netSeqNo(pkt) := lsno$

end

Event *start_tx_routePkt* $\widehat{=}$

//(ctl) starting transmiting a route packet to a channel

extends *start_tx_routePkt*

any

x

pkt

nbh

$lsno$

where

g1 : $x \mapsto pkt \in ndBuff$

g2 : $x \mapsto pkt \notin sentDown$

g3 : $x \mapsto pkt \notin sentUp$

g4 : $pkt \in dom(pktFwdr)$

g5 : $x \mapsto \{pkt \mapsto nbh\} \in nbHops$
g6 : $pkt \in dom(pktNbHops)$
g7 : $type(pkt) \neq BEACON$
g8 : $lsno \in \mathbb{N}$
g9 : $type(pkt) = ROUTE$
g10 : $x \in dom(routeSeqNo) \wedge lsno = routeSeqNo(x) + 1$

then

a1 : $ndBuff := ndBuff \setminus \{x \mapsto pkt\}$
a2 : $sentDown := sentDown \cup \{x \mapsto pkt\}$
a3 : $pktFwdr := pktFwdr \lhd\!\!\!- \{pkt \mapsto x\}$
a4 : $pktNbHops := pktNbHops \lhd\!\!\!- \{pkt \mapsto nbh\}$
a5 : $nbHops := nbHops \setminus \{x \mapsto \{pkt \mapsto nbh\}\}$
a6 : $netSeqNo(pkt) := lsno$
a7 : $routeSeqNo := routeSeqNo \lhd\!\!\!- \{x \mapsto lsno\}$

end

Event *send_down* $\widehat{=}$
//(shd) sending a packet down to a channel
extends *send_down*

any

cn
pkt
sno
src
$data$
$fwdr$
nbh
$dest$

where

g1 : $cn \mapsto pkt \in sentDown$
g2 : $cn \mapsto pkt \notin channel$
g3 : $pkt \in dom(pktSeqNo) \wedge sno = pktSeqNo(pkt)$
g4 : $pkt \in dom(pktSrc) \wedge src = pktSrc(pkt)$
g5 : $pkt \in dom(pktFwdr) \wedge fwdr = pktFwdr(pkt)$
g6 : $pkt \in dom(pktData) \wedge data = pktData(pkt)$
g7 : $pkt \in dom(pktNbHops) \wedge nbh = pktNbHops(pkt)$
g8 : $pkt \notin dom(vPktSeqNo)$
g9 : $pkt \notin dom(vPktFwdr)$
g10 : $pkt \notin dom(vPktSrc)$
g11 : $pkt \notin dom(vPktData)$
g12 : $pkt \notin dom(vPktNbHops)$
g13 : $pkt \in dom(pktDestAddr) \wedge dest = pktDestAddr(pkt)$

g14 : $pkt \notin dom(vPktDestAddr) \wedge dest \in ND \cup \{BROADCAST\}$

then

a1 : $channel := channel \cup \{cn \mapsto pkt\}$

a2 : $vPktSeqNo := vPktSeqNo \cup \{pkt \mapsto sno\}$

a3 : $vPktSrc := vPktSrc \cup \{pkt \mapsto src\}$

a4 : $vPktFwdr := vPktFwdr \cup \{pkt \mapsto fwdr\}$

a5 : $vPktData := vPktData \cup \{pkt \mapsto data\}$

a6 : $vPktNbHops := vPktNbHops \cup \{pkt \mapsto nbh\}$

a7 : $pktSeqNo := \{pkt\} \lhd pktSeqNo$

a8 : $pktSrc := \{pkt\} \lhd pktSrc$

a9 : $pktFwdr := \{pkt\} \lhd pktFwdr$

a10 : $pktData := \{pkt\} \lhd pktData$

a11 : $pktNbHops := \{pkt\} \lhd pktNbHops$

a14 : $pktDestAddr := \{pkt\} \lhd pktDestAddr$

a15 : $vPktDestAddr := vPktDestAddr \cup \{pkt \mapsto dest\}$

end

Event *find_neighbours* $\widehat{=}$

//(env) channel finds the neighbour list of the forwarder

extends *find_neighbour*

any

s

nbs

f

pkt

$dest$

where

g1 : $channel \neq \varnothing$

g2 : $pkt \in ran(channel)$

g3 : $s = initialSrcAddr(pkt)$

g4 : $f \mapsto pkt \in channel$

g5 : $nbs \subseteq ND \wedge nbs = wsnLinks[\{f\}] \wedge nbs \neq \varnothing$

g6 : $envNeighbours = \varnothing$

g7 : $pkt \in dom(vPktDestAddr) \wedge dest = vPktDestAddr(pkt)$

g8 : $dest = BROADCAST$

then

a1 : $envNeighbours := envNeighbours \cup (\{pkt\} \times nbs)$

end

Event *assign_forwarder* $\widehat{=}$

//(env) channel assigns specific forwarder

extends *assign_forwarder*

any

s

nb

pkt

f

nbs

$dest$

where

g1 : $channel \neq \varnothing$

g2 : $pkt \in ran(channel)$

g3 : $s = initialSrcAddr(pkt)$

g4 : $f \mapsto pkt \in channel$

g5 : $nbs = wsnLinks[\{f\}] \wedge nbs \neq \varnothing$

g6 : $nb \in nbs$

g7 : $pkt \in dom(vPktDestAddr) \wedge dest = vPktDestAddr(pkt)$

g8 : $dest \neq BROADCAST$

g9 : $nb = dest$

then

a1 : $envNeighbours := envNeighbours \cup \{pkt \mapsto nb\}$

end

Event *send_up* $\widehat{=}$

//(shd) channel sends the neighbour information

extends *send_up*

any

pkt

$nbrs$

f

sno

src

$fwdr$

$data$

nbh

$dest$

where

g1 : $pkt \in ran(sentDown)$

g2 : $f \mapsto pkt \in sentDown$

g3 : $nbrs \in \{n | n \in (\mathbb{P}(ND)) \cup \{\{FAILED\_XMIT\}\}\}$

g4 : $nbrs \neq \varnothing$

g5 : $ran(\{pkt\} \lhd envNeighbours) \in (\mathbb{P}(ND) \cup \{\{FAILED\_XMIT\}\})$

g6 : $nbrs = ran(\{pkt\} \lhd envNeighbours) \wedge nbrs \neq \varnothing$

g7 : $pkt \notin dom(ctlNeighbours)$

g8 : $f \mapsto pkt \notin sentUp$

g9 : $f \mapsto pkt \in channel$
g10 : $pkt \in dom(vPktSeqNo) \wedge sno = vPktSeqNo(pkt)$
g11 : $pkt \in dom(vPktSrc) \wedge src = vPktSrc(pkt)$
g12 : $pkt \in dom(vPktFwdr) \wedge fwdr = vPktFwdr(pkt)$
g13 : $pkt \in dom(vPktData) \wedge data = vPktData(pkt)$
g14 : $pkt \in dom(vPktNbHops) \wedge nbh = vPktNbHops(pkt)$
g15 : $pkt \notin dom(pktSeqNo)$
g16 : $pkt \notin dom(pktSrc)$
g17 : $pkt \notin dom(pktFwdr)$
g18 : $pkt \notin dom(pktData)$
g19 : $pkt \notin dom(pktNbHops)$
g20 : $pkt \in dom(vPktDestAddr) \wedge dest = vPktDestAddr(pkt)$
g21 : $pkt \notin dom(pktDestAddr)$

then

a1 : $ctlNeighbours := ctlNeighbours \cup (\{pkt\} \times nbrs)$
a2 : $envNeighbours := \varnothing$
a3 : $channel := channel \setminus \{f \mapsto pkt\}$
a4 : $sentDown := sentDown \setminus \{f \mapsto pkt\}$
a5 : $sentUp := sentUp \cup \{f \mapsto pkt\}$
a6 : $pktSeqNo := pktSeqNo \cup \{pkt \mapsto sno\}$
a7 : $pktSrc := pktSrc \cup \{pkt \mapsto src\}$
a8 : $pktFwdr := pktFwdr \cup \{pkt \mapsto fwdr\}$
a9 : $pktData := pktData \cup \{pkt \mapsto data\}$
a10 : $pktNbHops := pktNbHops \cup \{pkt \mapsto nbh\}$
a11 : $vPktSeqNo := \{pkt\} \lhd vPktSeqNo$
a12 : $vPktSrc := \{pkt\} \lhd vPktSrc$
a13 : $vPktFwdr := \{pkt\} \lhd vPktFwdr$
a14 : $vPktData := \{pkt\} \lhd vPktData$
a15 : $vPktNbHops := \{pkt\} \lhd vPktNbHops$
a16 : $vPktDestAddr := vPktDestAddr \setminus \{pkt \mapsto dest\}$
a17 : $pktDestAddr := pktDestAddr \cup \{pkt \mapsto dest\}$

end

Event *receive_dataPkt* $\widehat{=}$

//(ctl) each neighbour node receives a data packet from a channel

extends *receive_dataPkt*

any

s
nb
pkt
nbh

where

g1 : $pkt \notin ran(sentDown) \wedge pkt \in ran(sentUp)$
g2 : $pkt \mapsto nb \in ctlNeighbours$
g3 : $nb \mapsto pkt \notin ndBuff$
g4 : $nb \mapsto pkt \notin (sentUp \cup sentDown)$
g5 : $s = initialSrcAddr(pkt) \wedge s \neq nb$
g6 : $nb \neq Sink$
g7 : $nb \in dom(floodTbl) \wedge pkt \notin floodTbl(nb)$
g8 : $pkt \in dom(pktNbHops) \wedge nbh = pktNbHops(pkt) + 1$
g9 : $type(pkt) = DATA$

then

a1 : $ndBuff := ndBuff \cup \{nb \mapsto pkt\}$
a2 : $ctlNeighbours := ctlNeighbours \setminus \{pkt \mapsto nb\}$
a3 : $floodTbl(nb) := floodTbl(nb) \cup \{pkt\}$
a4 : $nbHops := nbHops \cup \{nb \mapsto \{pkt \mapsto nbh\}\}$

end

Event *receive_dup_dataPkt* $\widehat{=}$

//(ctl) each neighbour node including sink receives a duplicated data packet from a channel

extends *receive_dup_dataPkt*

any

nb
pkt

where

g1 : $pkt \in dom(ctlNeighbours)$
g2 : $pkt \mapsto nb \in ctlNeighbours$
g3 : $nb \in dom(floodTbl) \wedge pkt \in floodTbl(nb)$
g31 : $type(pkt) = DATA$

then

a1 : $ctlNeighbours := ctlNeighbours \setminus \{pkt \mapsto nb\}$

end

Event *receive_controlPkt* $\widehat{=}$

//(ctl) each neighbour node receives a control packet from a channel

extends *receive_controlPkt*

any

s
nb
pkt
nbh
f

where

g1 : $pkt \notin ran(sentDown) \wedge pkt \in ran(sentUp)$
g2 : $pkt \mapsto nb \in ctlNeighbours$
g3 : $nb \mapsto pkt \notin ndBuff$
g4 : $nb \mapsto pkt \notin (sentUp \cup sentDown)$
g5 : $s = initialSrcAddr(pkt) \wedge s \neq nb$
g6 : $nb \neq Sink$
g7 : $nb \in dom(floodTbl) \wedge pkt \notin floodTbl(nb)$
g8 : $pkt \in dom(pktNbHops) \wedge nbh = pktNbHops(pkt) + 1$
g9 : $type(pkt) \in CONTROL$
g10 : $pkt \in dom(pktFwdr) \wedge f = pktFwdr(pkt)$
g11 : $f \in ND \wedge nb \in ND$
g12 : $f \mapsto nb \notin updateNbrs$

then

a1 : $ndBuff := ndBuff \cup \{nb \mapsto pkt\}$
a3 : $ctlNeighbours := ctlNeighbours \setminus \{pkt \mapsto nb\}$
a4 : $floodTbl(nb) := floodTbl(nb) \cup \{pkt\}$
a5 : $nbHops := nbHops \cup \{nb \mapsto \{pkt \mapsto nbh\}\}$
a6 : $updateNbrs := updateNbrs \cup \{f \mapsto nb\}$

end

Event *receive_dup_controlPkt* $\widehat{=}$

//ctl) each neighbour node including sink receives a duplicated control packet from a channel

extends *receive_dup_controlPkt*

any

nb
pkt
f

where

g1 : $pkt \in dom(ctlNeighbours)$
g2 : $pkt \mapsto nb \in ctlNeighbours$
g3 : $nb \in dom(floodTbl) \wedge pkt \in floodTbl(nb)$
g9 : $type(pkt) \in CONTROL$
g11 : $pkt \in dom(pktFwdr) \wedge f = pktFwdr(pkt)$
g12 : $f \in ND \wedge nb \in ND$
g13 : $f \mapsto nb \notin updateNbrs$

then

a1 : $ctlNeighbours := ctlNeighbours \setminus \{pkt \mapsto nb\}$
a2 : $updateNbrs := updateNbrs \cup \{f \mapsto nb\}$

end

Event *sink_recv_dataPkt* $\widehat{=}$

// (ctl) each neighbour node receives a data pacekt from a channel

extends *sink_recv_dataPkt*

any

s

pkt

nb

where

g1 : $s = initialSrcAddr(pkt)$

g2 : $s \neq Sink$

g3 : $pkt \in ran(sentDown \cup sentUp) \setminus sinkBuff$

g4 : $nb = Sink$

g5 : $pkt \mapsto nb \in ctlNeighbours$

g6 : $nb \in dom(floodTbl) \wedge pkt \notin floodTbl(nb)$

g7 : $type(pkt) = DATA$

then

a1 : $sinkBuff := sinkBuff \cup \{pkt\}$

a2 : $ctlNeighbours := ctlNeighbours \setminus \{pkt \mapsto nb\}$

a3 : $floodTbl(nb) := floodTbl(nb) \cup \{pkt\}$

end

Event *sink_recv_controlPkt* $\widehat{=}$

//(ctl) sink receives a control packet

extends *scv_controlPkt*

any

s

pkt

nb

f

where

g1 : $s = initialSrcAddr(pkt)$

g2 : $s \neq Sink$

g3 : $pkt \in ran(sentDown \cup sentUp) \setminus sinkBuff$

g4 : $nb = Sink$

g5 : $pkt \mapsto nb \in ctlNeighbours$

g6 : $nb \in dom(floodTbl) \wedge pkt \notin floodTbl(nb)$

g7 : $type(pkt) \neq DATA$

g8 : $pkt \in dom(pktFwdr) \wedge f = pktFwdr(pkt)$

g9 : $f \in ND \wedge nb \in ND$

g10 : $f \mapsto nb \notin updateNbrs$

then

a1 : $sinkBuff := sinkBuff \cup \{pkt\}$

a2 : $ctlNeighbours := ctlNeighbours \setminus \{pkt \mapsto nb\}$

a3 : $floodTbl(nb) := floodTbl(nb) \cup \{pkt\}$

a4 : $updateNbrs := updateNbrs \cup \{f \mapsto nb\}$

end

Event *add_newEntry* $\widehat{=}$

//(ctl) adding new entry into the neighbour table

extends *add_newEntry*

any

x

y

where

g1 : $updateNbrs \neq \varnothing$

g2 : $x \in ND \wedge y \in ND \wedge x \mapsto y \in updateNbrs$

g4 : $y \mapsto x \notin neighbourTbl$

g5 : $y \mapsto x \mapsto 0 \notin missed$

g6 : $y \mapsto x \mapsto 0 \notin received$

g7 : $y \mapsto x \mapsto 0 \notin lastSeqno$

g8 : $y \mapsto x \mapsto 0 \notin receiveEst$

g9 : $y \mapsto x \mapsto 0 \notin sentEst$

g10 : $y \mapsto x \mapsto LIVELINESS \notin liveliness$

g11 : $y \mapsto x \mapsto BROADCAST \notin parent$

g12 : $(y \mapsto x \mapsto MAX\_DIST \notin cost) \wedge (x \neq Sink)$

then

a1 : $neighbourTbl := neighbourTbl \cup \{y \mapsto x\}$

a2 : $missed := missed \cup \{y \mapsto x \mapsto 0\}$

a3 : $received := received \cup \{y \mapsto x \mapsto 0\}$

a4 : $lastSeqno := lastSeqno \cup \{y \mapsto x \mapsto 0\}$

a5 : $receiveEst := receiveEst \cup \{y \mapsto x \mapsto 0\}$

a6 : $sentEst := sentEst \cup \{y \mapsto x \mapsto 0\}$

a7 : $liveliness := liveliness \cup \{y \mapsto x \mapsto LIVELINESS\}$

a8 : $parent := parent \cup \{y \mapsto x \mapsto BROADCAST\}$

a9 : $cost := cost \cup \{y \mapsto x \mapsto MAX\_DIST\}$

end

Event *add_newEntry_sink* $\widehat{=}$

//(ctl) adding new entry into the neighbour table

extends *add_newEntry*

any

x

y

where

g1 : $updateNbrs \neq \varnothing$

g2 : $x \in ND \wedge y \in ND \wedge x \mapsto y \in updateNbrs$

g4 : $y \mapsto x \notin neighbourTbl$
g5 : $y \mapsto x \mapsto 0 \notin missed$
g6 : $y \mapsto x \mapsto 0 \notin received$
g7 : $y \mapsto x \mapsto 0 \notin lastSeqno$
g8 : $y \mapsto x \mapsto 0 \notin receiveEst$
g9 : $y \mapsto x \mapsto 0 \notin sentEst$
g10 : $y \mapsto x \mapsto LIVELINESS \notin liveliness$
g11 : $y \mapsto x \mapsto BROADCAST \notin parent$
g12 : $(y \mapsto x \mapsto 0 \notin cost) \wedge (x = Sink)$

then

a1 : $neighbourTbl := neighbourTbl \cup \{y \mapsto x\}$
a2 : $missed := missed \cup \{y \mapsto x \mapsto 0\}$
a3 : $received := received \cup \{y \mapsto x \mapsto 0\}$
a4 : $lastSeqno := lastSeqno \cup \{y \mapsto x \mapsto 0\}$
a5 : $receiveEst := receiveEst \cup \{y \mapsto x \mapsto 0\}$
a6 : $sentEst := sentEst \cup \{y \mapsto x \mapsto 0\}$
a7 : $liveliness := liveliness \cup \{y \mapsto x \mapsto LIVELINESS\}$
a8 : $parent := parent \cup \{y \mapsto x \mapsto BROADCAST\}$
a9 : $cost := cost \cup \{y \mapsto x \mapsto 0\}$

end

Event *update_nbr* $\widehat{=}$

//(ctl) updating received information into the neighbour table

extends *update_nbr*

any

x
y
sNo
$delta$
pkt

where

g1 : $updateNbrs \neq \varnothing$
g2 : $x \in ND \wedge y \in ND \wedge x \mapsto y \in updateNbrs$
g3 : $y \mapsto x \in neighbourTbl$
g4 : $pkt \mapsto x \in pktFwdr$
g5 : $sNo = netSeqNo(pkt)$
g6 : $delta = sNo - lastSeqno(y \mapsto x) - 1$
g7 : $delta \geq 0$
g8 : $type(pkt) = BEACON$
g9 : $pkt \in ran(sentUp)$

then

a1 : $received := received \Leftarrow \{y \mapsto x \mapsto (received(y \mapsto x) + 1)\}$

a2 : $lastSeqno := lastSeqno \lhd\!\!\!+ \{y \mapsto x \mapsto sNo\}$
a3 : $missed := missed \lhd\!\!\!+ \{y \mapsto x \mapsto delta\}$
a4 : $updateNbrs := updateNbrs \setminus \{x \mapsto y\}$

end

Event *update_route_1* $\widehat{=}$

//(ctl) updating information from route packet into the neighbour table (for sensor nodes)

extends *update_route*

any

x
y
pkt
par
cst

where

g1 : $updateNbrs \neq \varnothing$
g2 : $x \in ND \wedge y \in ND \wedge x \mapsto y \in updateNbrs$
g3 : $y \mapsto x \in neighbourTbl$
g4 : $pkt \mapsto x \in pktFwdr$
g5 : $type(pkt) \neq DATA \wedge type(pkt) \neq BEACON$
g6 : $pkt \in ran(sentUp)$
g7 : $y \mapsto x \in dom(sentEst)$
g8 : $x \mapsto y \in dom(receiveEst)$
g9 : $y \mapsto x \in dom(liveliness)$
g10 : $x \in dom(cRouteTree) \wedge par = cRouteTree(x)$
g11 : $x \in dom(cpCost) \wedge cst = cpCost(x) \wedge cst \neq MAX\_DIST$
g12 : $y \mapsto x \in dom(parent) \wedge y \mapsto x \mapsto par \notin parent$
g13 : $y \mapsto x \in dom(cost) \wedge y \mapsto x \mapsto cst \notin cost$

then

a1 : $updateNbrs := updateNbrs \setminus \{x \mapsto y\}$
a2 : $sentEst(y \mapsto x) := receiveEst(x \mapsto y)$
a3 : $liveliness(y \mapsto x) := LIVELINESS$
a4 : $parent := parent \lhd\!\!\!+ \{y \mapsto x \mapsto par\}$
a5 : $cost := cost \lhd\!\!\!+ \{y \mapsto x \mapsto cst\}$

end

Event *update_route_2* $\widehat{=}$

//(ctl) updating information from route packet into the neighbour table (for a sink)

extends *update_route*

any

x

y

pkt

where

g1 : $updateNbrs \neq \varnothing$

g2 : $x \in ND \wedge y \in ND \wedge x \mapsto y \in updateNbrs$

g3 : $y \mapsto x \in neighbourTbl$

g4 : $pkt \mapsto x \in pktFwdr$

g5 : $type(pkt) \neq DATA \wedge type(pkt) \neq BEACON$

g6 : $pkt \in ran(sentUp)$

g7 : $y \mapsto x \in dom(sentEst)$

g8 : $x \mapsto y \in dom(receiveEst)$

g9 : $y \mapsto x \in dom(liveliness)$

g10 : $x \notin dom(cRouteTree)$

then

a1 : $updateNbrs := updateNbrs \setminus \{x \mapsto y\}$

a2 : $sentEst(y \mapsto x) := receiveEst(x \mapsto y)$

a3 : $liveliness(y \mapsto x) := LIVELINESS$

end

Event *decrease_liveliness* $\widehat{=}$

//(ctl) decreasing liveliness of each neighbour in neighbour table

extends *decrease_liveliness*

any

x

y

$live$

where

g1 : $estNbrs = neighbourTbl$

g2 : $x \mapsto y \in checkedLiveNbrs$

g3 : $x \mapsto y \in dom(liveliness)$

g4 : $live = liveliness(x \mapsto y) \wedge live > 0$

then

a1 : $liveliness := liveliness \lhd\!\!\!+ \{x \mapsto y \mapsto (live - 1)\}$

a2 : $checkedLiveNbrs := checkedLiveNbrs \setminus \{x \mapsto y\}$

end

Event *exclude_deadNbr* $\widehat{=}$

//(ctl) excluding dead neighbour from neighbour table

extends *exclude_deadNbr*

any

x

y

$live$

where

g1 : $estNbrs = neighbourTbl$

g2 : $x \mapsto y \in checkedLiveNbrs$

g3 : $x \mapsto y \in dom(liveliness)$

g4 : $live = liveliness(x \mapsto y) \wedge live = 0$

g5 : $y \mapsto x \in dom(sentEst)$

then

a1 : $sentEst(y \mapsto x) := 1$

a2 : $checkedLiveNbrs := checkedLiveNbrs \setminus \{x \mapsto y\}$

end

Event *est_due* $\widehat{=}$

//(ctl) link quality estimation timer due

extends *est_due*

any

x

where

g1 : $x \in ND \setminus estNDs$

g2 : $x \in dom(totalSentBcon) \wedge totalSentBcon(x) > 0$

g3 : $totalSentBcon(x) mod EST\_RATIO = 0$

then

a1 : $estNDs := estNDs \cup \{x\}$

end

Event *update_est* $\widehat{=}$

//(ctl) calculate and update reception ratio in case no lost packets or lose packets in the middle of transmission

extends *update_est*

any

nd

nb

r

m

rt

$newAve$

where

g1 : $nd \in estNDs$

g2 : $nb \in ran(\{nd\} \lhd neighbourTbl)$

g3 : $nd \in dom(totalSentBcon) \wedge totalSentBcon(nd) > 0$

g4 : $nd \mapsto nb \in dom(lastSeqno) \wedge lastSeqno(nd \mapsto nb) > 0$

g5 : $r = received(nd \mapsto nb)$

g6 : $m = missed(nd \mapsto nb)$
g8 : $rt = EST\_RATIO$
g9 : $newAve = ((r * 100)/rt)$
g10 : $nd \mapsto nb \in dom(receiveEst) \wedge newAve > 0$
g11 : $nd \mapsto nb \in dom(missed)$
g12 : $nd \mapsto nb \in dom(received)$
g13 : $nd \mapsto nb \notin estNbrs$
g14 : $estNbrs \neq neighbourTbl$
g15 : $nd \mapsto nb \notin checkedLiveNbrs$

then

a1 : $receiveEst := receiveEst \Leftarrow \{nd \mapsto nb \mapsto newAve\}$
a2 : $missed := missed \Leftarrow \{nd \mapsto nb \mapsto 0\}$
a3 : $received := received \Leftarrow \{nd \mapsto nb \mapsto 0\}$
a4 : $estNbrs := estNbrs \cup \{nd \mapsto nb\}$
a7 : $checkedLiveNbrs := checkedLiveNbrs \cup \{nd \mapsto nb\}$

end

Event *update_est_nothing* $\widehat{=}$

//(ctl) for case the dead neighbour node affecting received become zero

extends *update_est_nothing*

any

nd
nb
r
m
rt
$newAve$

where

g1 : $nd \in estNDs$
g2 : $nb \in ran(\{nd\} \lhd neighbourTbl)$
g3 : $nd \in dom(totalSentBcon) \wedge totalSentBcon(nd) > 0$
g4 : $nd \mapsto nb \in dom(lastSeqno) \wedge lastSeqno(nd \mapsto nb) > 0$
g5 : $r = received(nd \mapsto nb)$
g6 : $m = missed(nd \mapsto nb)$
g7 : $rt = EST\_RATIO \wedge rt > 0$
g8 : $newAve = ((r * 100)/rt)$
g9 : $newAve = 0$
g10 : $nd \mapsto nb \notin estNbrs$
g11 : $estNbrs \neq neighbourTbl$
g12 : $nd \mapsto nb \notin checkedLiveNbrs$

then

a1 : $estNbrs := estNbrs \cup \{nd \mapsto nb\}$

a2 : $checkedLiveNbrs := checkedLiveNbrs \cup \{nd \mapsto nb\}$

end

Event *finish_tx_pkt* $\widehat{=}$

//(ctl) finishing packet transmission

extends *finish_tx_pkt*

any

f

pkt

where

g1 : $\{pkt\} \lhd ctlNeighbours = \varnothing$

g2 : $pkt \in ran(sentUp) \wedge pkt \notin ran(sentDown) \wedge pkt \in ran(ndBuff)$

g3 : $f \in dom(sentUp) \wedge f \mapsto pkt \in sentUp$

g4 : $type(pkt) = DATA \vee type(pkt) = BEACON$

then

a1 : $sentUp := sentUp \setminus \{f \mapsto pkt\}$

end

Event *final_tx_pkt* $\widehat{=}$

//(ctl) the transmitted packet is removed from middleware indicating completion of transmission mechanism

extends *final_tx_pkt*

any

pkt

where

g1 : $\{pkt\} \lhd ctlNeighbours = \varnothing$

g2 : $pkt \in ran(sentUp) \wedge pkt \notin ran(sentDown) \wedge pkt \notin ran(ndBuff)$

g41 : $type(pkt) = DATA \vee type(pkt) = BEACON$

then

a1 : $sentUp := sentUp \rhd \{pkt\}$

end

Event *chooseParent_due* $\widehat{=}$

//(ctl) choosing a parent timer due

extends *reset_bcastRouTimer*

when

g1 : $bcastRouTimer = TRUE$

g2 : $bcastRouNodes = ND$ //when all node broadcast the route packet

g3 : $chooseParentTimer = FALSE$

then

a1 : $bcastRouTimer := FALSE$

a2 : $bcastRouNodes := \varnothing$

a3 : $chooseParentTimer := TRUE$

end

Event *choose_node* $\widehat{=}$

//choosing a node who has not chosen its parent yet

any

x

nb

where

g1 : $chooseParentTimer = TRUE$

g2 : $x \in ND \setminus \{Sink\}$

g3 : $x \notin chosenNDs$

g4 : $nb = ran(\{x\} \lhd neighbourTbl) \wedge nb \neq \varnothing$

g5 : $Nbrs = \varnothing$

g6 : $completedRoute = FALSE$

g7 : $calPCostFlg = FALSE$

then

a1 : $chosenNDs := chosenNDs \cup \{x\}$

a2 : $Nbrs := nb$

end

Event *cal_PCost_gt* $\widehat{=}$

//(ctl) calculating path cost

any

x

nb

$pCost$

$Nbrs1$

$rEst$

$sEst$

$lnkQty$

th

where

g1 : $x \in chosenNDs \wedge nb \in Nbrs$

g2 : $th \in QThreshold$

g3 : $th > 0 \wedge th = Qlow$

g4 : $x \mapsto nb \in neighbourTbl$

g5 : $x \mapsto nb \in dom(receiveEst) \wedge receiveEst(x \mapsto nb) \geq th$

g6 : $rEst \in \mathbb{N} \wedge rEst = receiveEst(x \mapsto nb)$

g7 : $x \mapsto nb \in dom(sentEst) \wedge sentEst(x \mapsto nb) \geq th$

g8 : $sEst \in \mathbb{N} \wedge sEst = sentEst(x \mapsto nb)$

g9 : $lnkQty \in \mathbb{N} \wedge lnkQty = ((120 - rEst) * (120 - sEst))$

g10 : $Nbrs1 = ran(\{x\} \lhd neighbourTbl) \wedge Nbrs1 \neq \varnothing$

g11 : $pCost \in \mathbb{N}$

g12 : $cpCost(nb) = pCost \wedge pCost \neq MAX\_DIST$
g13 : $x \mapsto nb \notin calPCostNDs$
g14 : $pCost + lnkQty \geq 0$

then

a1 : $calPCostNDs := calPCostNDs \cup \{x \mapsto nb\}$
a2 : $calPCost := calPCost \cup \{x \mapsto nb \mapsto (pCost + lnkQty)\}$
a3 : $Nbrs := Nbrs \setminus \{nb\}$
a4 : $calPCostFlg := TRUE$

end

Event *cal_PCost_gt_skip* $\widehat{=}$

//(ctl) skipping calculating path cost for the case that the node is further far away from the sink

any

x
nb
$lnkQty$
$Nbrs1$
$pREst$
$pSEst$
$pCost$
th

where

g1 : $x \in chosenNDs$
g2 : $th \in QThreshold$
g3 : $th > 0 \wedge th = Qlow$
g4 : $nb \in Nbrs$
g5 : $x \mapsto nb \in neighbourTbl$
g6 : $x \mapsto nb \in dom(receiveEst) \wedge receiveEst(x \mapsto nb) \geq th$
g7 : $pREst \in \mathbb{N} \wedge pREst = receiveEst(x \mapsto nb)$
g8 : $x \mapsto nb \in dom(sentEst) \wedge sentEst(x \mapsto nb) \geq th$
g9 : $pSEst \in \mathbb{N} \wedge pSEst = sentEst(x \mapsto nb)$
g10 : $lnkQty \in \mathbb{N} \wedge lnkQty = ((120 - pREst) * (120 - pSEst))$
g11 : $Nbrs1 = ran(\{x\} \lhd neighbourTbl) \wedge Nbrs1 \neq \varnothing$
g12 : $pCost \in \mathbb{N}$
g13 : $cpCost(nb) = pCost \wedge pCost = MAX\_DIST$
g14 : $x \mapsto nb \notin calPCostNDs$
g15 : $pCost + lnkQty \geq 0$

then

a1 : $Nbrs := Nbrs \setminus \{nb\}$
a2 : $calPCostFlg := TRUE$

end

Event *cal_PCost_le* $\widehat{=}$

//(ctl) skipping calculating path cost for the case that the neighbour node has the link quality less than the threshold

any

x

nb

$pREst$

th

where

g1 : $x \in chosenNDs$

g2 : $th \in QThreshold$

g2 : $th > 0 \wedge th = Qlow$

g3 : $nb \in Nbrs$

g5 : $x \mapsto nb \in neighbourTbl$

g6 : $x \mapsto nb \in dom(receiveEst) \wedge x \mapsto nb \in dom(sentEst)$

g7 : $pREst \in \mathbb{N} \wedge pREst = receiveEst(x \mapsto nb)$

g8 : $receiveEst(x \mapsto nb) \geq th \wedge sentEst(x \mapsto nb) \geq th$

g8 : $receiveEst(x \mapsto nb) < th \vee sentEst(x \mapsto nb) < th$

then

a1 : $Nbrs := Nbrs \setminus \{nb\}$

a2 : $calPCostFlg := TRUE$

end

Event *choose_parent* $\widehat{=}$

//(ctl) choosing a parent

any

x

p

m

where

g1 : $p \in ND$

g2 : $x \in nodes \setminus \{Sink\}$

g3 : $x \notin dom(cRouteTree)$

g4 : $completedRoute = FALSE$

g5 : $Nbrs = \varnothing$

g6 : $ran(calPCost) \neq \varnothing$

g7 : $m = min(ran(calPCost))$

g9 : $x \mapsto p \in calPCostNDs$

g10 : $calPCostFlg = TRUE$

g11 : $x \neq p$

g12 : $des \subseteq nodes \wedge (tcl(cRouteTree))^{-1}[\{p\}]$

g13 : $p \notin des \cup \{x\}$

then

a1 : $cRouteTree := cRouteTree \cup \{x \mapsto p\}$
a2 : $nodes := nodes \cup \{x\}$
a3 : $cpCost(x) := m$
a4 : $calPCostNDs := \varnothing$
a5 : $calPCost := \varnothing$
a6 : $calPCostFlg := FALSE$

end

Event *choose_parent_skip* $\widehat{=}$

//(ctl) skipping choosing parant for the case that all links are lower than the quality threshold

any

x
p
m

where

typing_p : $p \in \mathbb{Z}$
typing_m : $m \in \mathbb{Z}$
typing_x : $x \in \mathbb{Z}$
g1 : $x = x$
g2 : $p \in nodes$
g3 : $x \in ND \setminus \{Sink\}$
g4 : $x \notin dom(cRouteTree)$
g5 : $completedRoute = FALSE$
g6 : $Nbrs = \varnothing$
g7 : $calPCostNDs = \varnothing$
g8 : $calPCostFlg = TRUE$

then

a1 : $calPCostFlg := FALSE$

end

Event *report_routeCompletion* $\widehat{=}$

//(ctl) reporting the route completion

any

r

where

typing_r : $r \in \mathbb{P}(\mathbb{Z} \times \mathbb{Z})$
g1 : $completedRoute = FALSE$
g2 : $r = cRouteTree$
g3 : $nodes = ND$
g4 : $Nbrs = \varnothing$

then

a1 : $completedRoute := TRUE$

end

Event *reset_route* $\widehat{=}$

//(ctl) resetting the route tree

when

g1 : $completedRoute = TRUE$

g2 : $chooseParentTimer = TRUE$

g3 : $dom(cRouteTree) = ND \setminus \{Sink\}$

g4 : $chosenNDs \neq \varnothing$

then

a1 : $completedRoute := FALSE$

a2 : $cRouteTree := \varnothing$

a3 : $nodes := \{Sink\}$

a4 : $chosenNDs := \varnothing$

a5 : $chooseParentTimer := FALSE$

end

Event *lose_all_neighbours* $\widehat{=}$

//(env) channel loses all neighbuor

extends *lose_all_neighbours*

any

s

nbs

f

pkt

where

g1 : $channel \neq \varnothing$

g2 : $pkt \in ran(channel)$

g3 : $s = initialSrcAddr(pkt)$

g4 : $f \mapsto pkt \in channel$

g5 : $nbs = wsnLinks[\{f\}] \wedge nbs = \varnothing$

then

a1 : $envNeighbours := envNeighbours \cup \{pkt \mapsto FAILED\_XMIT\}$

end

Event *lose_pkt* $\widehat{=}$

//(ctl) each node records a lost packet

extends *lose_pkt*

any

f

pkt

where

$\texttt{g1} : pkt \in ran(sentUp) \wedge pkt \notin ran(sentDown) \wedge pkt \notin ran(ndBuff)$

$\texttt{g2} : pkt \notin ran(recLostPkts)$

$\texttt{g3} : f \mapsto pkt \in sentUp \wedge f \mapsto pkt \notin sentDown$

$\texttt{g4} : f \mapsto pkt \notin recLostPkts$

$\texttt{g5} : ran(\{pkt\} \lhd ctlNeighbours) = \{FAILED\_XMIT\}$

then

$\texttt{a1} : sentUp := sentUp \rhd\!\!\!- \{pkt\}$

$\texttt{a2} : recLostPkts := recLostPkts \cup \{f \mapsto pkt\}$

end

END

Appendix B

FoCoSim-WSN API Specification

This section describes FoCoSim-WSN API Specification. This extends from programmable API for ProB and MiXiM simulation frameworks. Therefore, this section only describes the APIs that are extended on these base frameworks. For further API reference, see ProB 2.0 Developer Handbook (Bendisposto and Clark, 2014) for ProB's APIs and OMNeT++ API Reference (OMNeT++, 2014) for MiXiM's APIs.

B.1 Master's API Reference

B.1.1 Overview of Master's API Reference

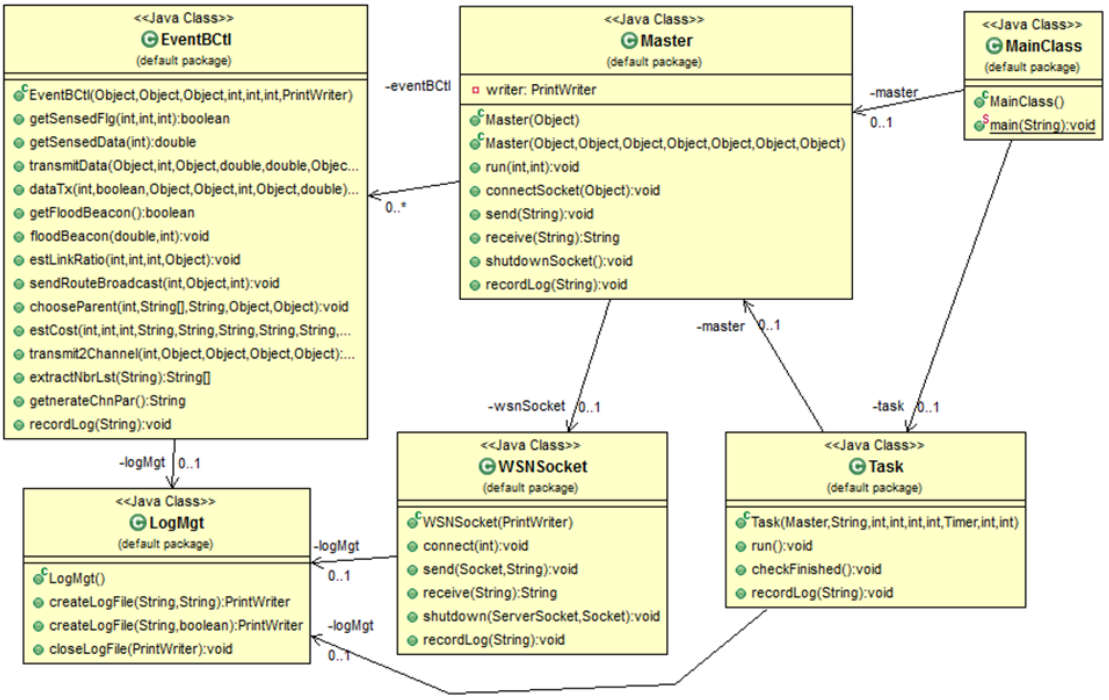
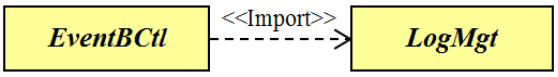


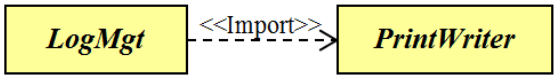
Figure B.1: Master structure implementation

B.1.1.1 Class *EventBCtl*

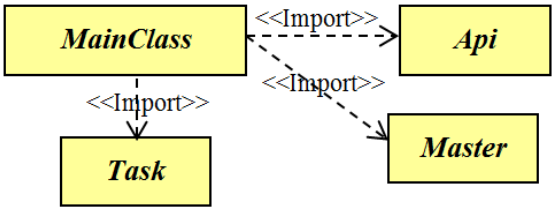
Detailed Description
The controller interface class encoding the event traces in Event-B controller including *SensorApp* and *MintRoute* model for *SensorScope* case study.
Diagram for *EventBCtl* 
Class *LogMgt* - see Section B.1.1.2
Constructor Summary
EventBCtl(def dCtl, def bCtl, def rCtl, int numNDs, int sinkAddr, int estRatio, PrintWriter writer) The Event-B controller model object instantiated by Api (*dCtl*-for data sensing mechanism, bCtl - for beacon flooding mechanism and *rCtl* - for route broadcasting mechanism), configuration parameters (*numNDs*, *SinkAddr*, *estRatio*) and logfile's writer pointer (*writer*) are injected into a task object at start up.
Method Summary
public boolean getSensedFlg(int nd, int port, int senseCount) Retrieves the sensed flag passed from simulation environment models and returns this boolean flag.
public double getSensedData(int nd) Retrieves the sensed data passed from simulation environment models and returns this sensed data.
public void transmitData(def sensingND, int port, def senseCount,double startTime, double sensedData, def completedRouteFlg, def cRouteTree) Prepares sensed data for being transmitted down to the channel.
private void dataTx(int numNDP, boolean completedRouteFlg, def cRouteTree, def nd, int port, def i, double sensedData,def senseCount) Creates a data packet containing sensed data. This method is called by method transmitData().
public boolean getFloodBeacon() Retrieves the beacon broadcast flag passed from simulation environment models. This boolean flag is returned from this method.
public void floodBeacon(double startTime,int tempCount) Starts flooding a beacon to the channel.
public void estCost(int estTick, int rRound, int nd) Starts estimating the link quality ratio.
private void estLinkRatio(int nd, int estTick, int rRound, def neighbourTbl) Calculating the link quality ratio. This method is called by method *estCost()*.
public void sendRouteBroadcast(int nd, def estTick,int rRound) Starts broadcasting a route update message to the channel.
public void chooseParent(int nd, String[] neighbourTbl, String totalSentBcon, int rRound,int ndLsts) Choosing a parent for node *nd*.
private void transmit2Channel(int nd, def ctl, def pktType, def port, def sensedData, def senseCount) Transmits the packet to the channel. This method is called by method *dataTx()*, *floodBeacon()* and *sendRouteBroadcast()*.

Method Summary
private void extractNbrLst(String nbrLst):String[] Extracting the neighbour list (*nbrLst*) passed by MiXiM environment.
private void generateChnPar():String Encoding the channel information parameter into xml format.

B.1.1.2 Class *LogMgt*

Detailed Description
Log file management class includes the creation, open, write, and close operation on a log file.
Diagram for *LogMgt* 
Class *PrintWriter* - This class prints formatted data to a text-output stream.
Method Summary
PrintWriter createLogFile(String name, String code) Creates a new log file with the specified name (name) and charset (code). PrinterWriter object is returned from this contructor.
PrintWriter createLogFile(String name, boolean autoFlush) Creates a new log file from an existing file. PrinterWriter object is returned from this contructor.
void closeLogFile(PrintWriter fileObj) Closes a log file handled by PrintWriter object.

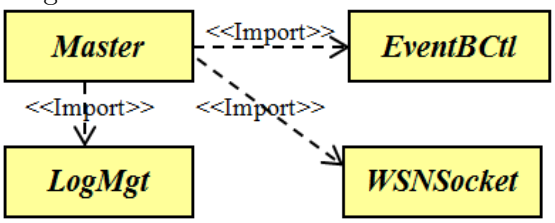
B.1.1.3 Class *MainClass*

Detailed Description
The main class to drive the master execution on ProB.
Diagram for *MainClass* 
Class *Api* - A ProB's API for retrieving the formal model elements e.g. states of variables and events. This includes for *ClassicalBModel*, *CSPModel* and *EeventBModel*.
Class *Master* - See Section B.1.1.4 for the detail.
Class *Task* See Section B.1.1.5 for the detail.
Method Summary
public static void main(String[] args) Loading Event-B models and running ProB simulation

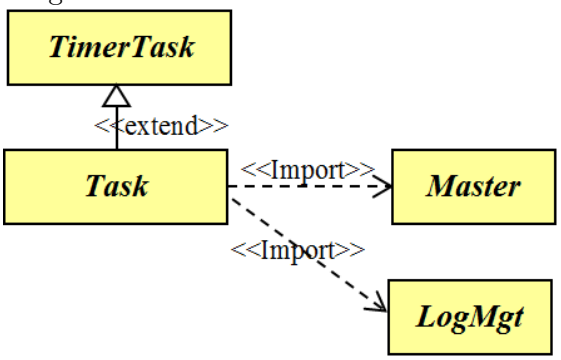
See more detail for Class *Api*[1]

[1] see Class *Api* - http://nightly.cobra.cs.uni-duesseldorf.de/prob2/javadoc/de/prob/scripting/Api.html

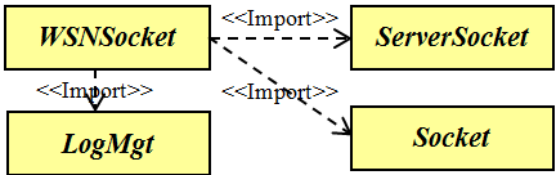
B.1.1.4 Class *Master*

Detailed Description
The master class for driving the co-simulation. The parameter exchanging between Event-B models and simulation environment is also provided in this class.
Diagram for *Master* 
Class *EventBCtl* - see Section B.1.1.1
Class *WSNSocket* - see Section B.1.1.6
Class *LogMgt* - see Section B.1.1.2
Constructor Summary
Master(EventBCtl dCtl, EventBCtl bCtl, EventBCtl rCtl, int numNDs, int sinkAddr,int estRatio, PrintWriter writer) The Event-B controller model object instantiated by *Api* (*dCtl*-for data sensing mechanism, *bCtl* - for beacon flooding mechanism and *rCtl* - for route broadcasting mechanism), configuration parameters (*numNDs*, *estRatio*, *SinkAddr*, *nd*) and logfile's writer pointer (*writer*) are injected into a task object at startup.
Method Summary
public void run(int threadType, int nd) Running a specific task depending on the *threadType*. This is defined into three categories: *data sensing*, *beacon flooding* and *route broadcasting mechanisms*.
public void connectSocket(int port) Establishing the connection to a socket.
public void send(String buffer) Sending input parameters extracted form input source (e.g. *EventBCtl* model) a to the socket.
public String receive(String msg) Receiving output parameters delivered from output destination (e.g. MiXiM environment models) via the socket.
public void shutdownSocket() Shutting down the socket.

B.1.1.5 Class *Task*

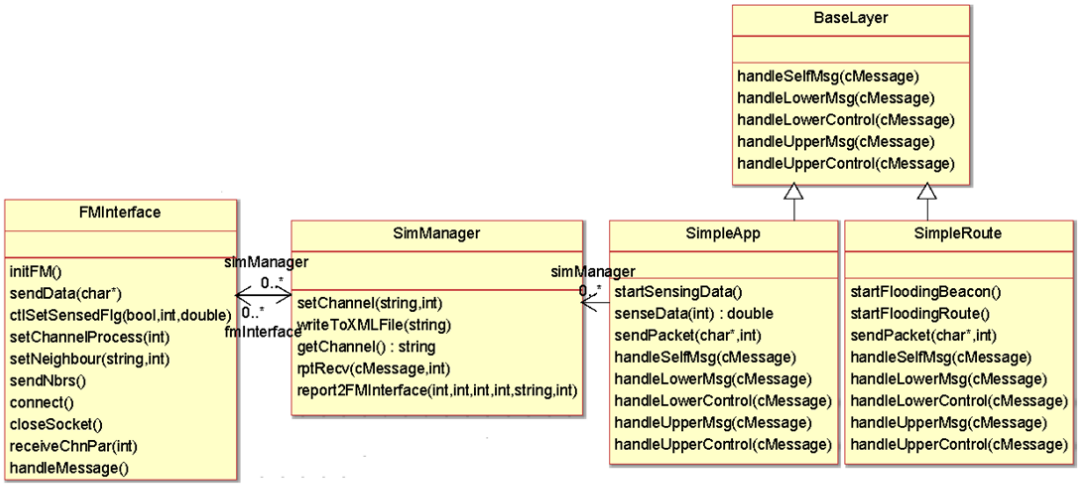
Detailed Description
The class for creating threads containing an instance of Event-B model and scheduling these threads periodically.
Diagram for *Task* 
Class *TimerTask* - The class creates instances (tasks) for a thread that can be scheduled for particular one-time or repeated execution managed by Timer. Class *Master* - see Section B.1.1.4 Class *LogMgt* - see Section B.1.1.2
Constructor Summary
Task(Master master, String threadName, int threadType, int numNDs, int estRatio, int SinkAddr, int nd, Timer timer, int repeat) The master object, thread information (*threadName* and *threadType*), configuration parameters (*numNDs*, *estRatio*, *SinkAddr*, *nd*) and timer information (*timer*, *repeat*) are injected into a task object at start up.
Method Summary
public void run() Running a Thread's task which is executing the master operation.
public void checkFinished() Checking timer's finishing point. If the current time is greater than repeat, then the thread terminates.

B.1.1.6 Class *WSNSocket*

Detailed Description
A socket interface class connect to MiXiM environment models.
Diagram for *WSNSocket* 
Class *ServerSocket* - This class implements server sockets. A server socket listens for client requests over the network. Then, it performs operation based on the received request, an then possibly returns a result to the client. Class *Socket* - This class implements client sockets. A socket is an endpoint for communication between two programs running on the network. Class *LogMgt* - see Section B.1.1.2
Constructor Summary
WSNSocket(PrintWriter writerP) Instantiates *PrintWriter* objects used for *LogMgt*
Method Summary
public void connect(int port) Creates a server socket and establishes the connection to the client.
public void send(Socket socket, String buffer) Sends a message stored in buffer to the destination via a specific socket.
public String receive(String msg) Receives a message from the destination. The content in the message is returned.
public void shutdown(ServerSocket server, Socket socket) Shutting down both server and client socket.

B.2 MiXiM's API Reference

B.2.1 Overview of MiXiM API Reference

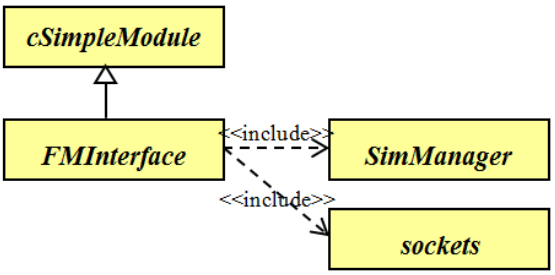


Classes
• *FMInterface*
• *SimManager*
• *SimpleApp*
• *SimpleRoute*

Figure B.2: MiXiM's extension implementation

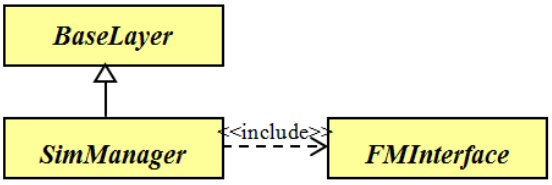
B.2.1.1 Class *FMInterface*

#include <FMInterface.h>

Detailed Description
The MiXiM's front-end interface class for synchronizing between formal controller models and physical environment models.
Diagram for *FMInterface* 
Class *cSimpleModule* - Base class for all simple module classes. Class *sockets* (platdep/sockets.h) - This class implements the TCP socket for parameter exchanger between co-models. Class *SimManager* - see Section B.2.1.2
Constructor Summary
virtual bool initFM() requests the connection to formal controller. The completion of the Event-B model initialisation in terms of boolean is returned by this function.
virtual void ctlSetSensedFlg(bool sensingFlg, int sensingNode, double sensedData) sets sensed flag with generating random sense data.
virtual void setNeighbour(std::string _neighbour, int _port) sets the neighbour list reported to the master.
virtual void chnSendNeighbour() sends the neighbour list to the master via the socket.
Protected Member Function Summary
virtual void sendData(const char contents)* sends the output parameter (the neighbour list) to the master by using the socket.
virtual void connect() Establishes the connection to a socket.
virtual void closeSocket() Closes and shutdowns the socket.
virtual void receiveChnPar(int id) receives the input parameter (packet information) from the master by using the socket.
*virtual void handleMessage(cMessage *msg)* indicating the synchronisation point for parameter exchange between formal controller and physical environment models.

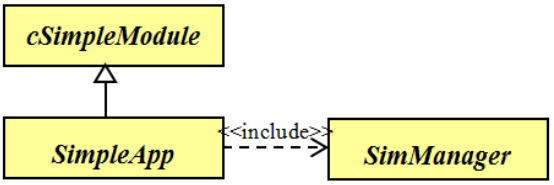
B.2.1.2 Class *SimManager*

#include <SimManager.h>

Detailed Description
The MiXiM's back-end interface class for passing between FMInterface to simple base modules layered on the base protocol stack.
Diagram for *SimManager* 
Class *BaseLayer* - Base class for all simple module classes. Class *FMInterface* - see Section B.2.1.1
Constructor and Destructor Summary
SimManager()
virtual ~SimManager()
Public Member Function Summary
virtual void setChannel(std::string var1, int totalnumNode) sets an input parameter (*chnPar*) passed from *FMInterface*.
virtual void writeToXMLFile(std::string contents) writes parameters in terms of XML format.
virtual std::string getChannel() retrieves the recorded input parameter (*chnPar*).
virtual void rptRecv(int nodeId, int cForwarder, int intSrcAddr,int cParent, std::string cPkt1, int cPktId, int seqNo,int NbHops, int data, int port, bool deadLnk) reports receiving packet information.
virtual bool report2FMInterface(int cForwarder, int cSrc, int cSeqno, int cParent, std::string cPktP, int port) reports the neighbour list to *FMInterface*.

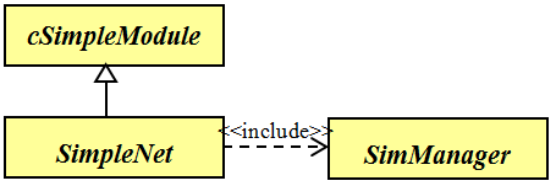
B.2.1.3 Class *SimpleApp*

#include <SimpleApp.h>

Detailed Description
The simple base class for base application layer.
Diagram for *SimpleApp* 
Class *cSimpleModule* - Base class for all simple module classes. Class *SimManager* - see Section B.2.1.2
Constructor and Destructor Summary
SimpleApp()
virtual ~SimpleApp()
Public Member Function Summary
virtual void startSensingData() indicates sensing data mechanism.
virtual double senseData(int src) generates sense data for the specific source node (*src*).
virtual void sendPacket(const char pktName, int pktType)* generates a virtual packet and transmits it down to the lower layer.
Inherited and Protected Member Function Summary
*virtual void handleSelfMsg(cMessage *)* implements the necessary timer for application protocol e.g. sensing timers by sending self-messages. Thus, this method responses to handle all timers and initiate action when timeouts.
*virtual void handleLowerMsg(cMessage *)* handles the message that has sent from an lower layer. After processing this messages, it can be forwarded to lower layers by using *sendUp*() function.
*virtual void handleLowerControl(cMessage * msg)* handles the control message that has sent from a lower layer.
*virtual void handleUpperMsg(cMessage * m)* handles the message that has sent from an upper layer. After processing this messages, it can be forwarded to lower layers by using *sendDown*() function.
*virtual void handleUpperControl(cMessage * m)* handles the control message that has sent from a upper layer.

B.2.1.4 Class *SimpleNet*

#include <SimpleNet.h>

Detailed Description
The simple base class for base application layer.
Diagram for *SimpleApp* 
Class *cSimpleModule* - Base class for all simple module classes. Class *SimManager* - see Section B.2.1.2
Constructor and Destructor Summary
SimpleNet()
virtual ~SimpleNet()
Public Member Function Summary
virtual void startFloodingBeacon() indicates flooding a beacon.
virtual void startFloodingRoute() indicates flooding a route packet.
virtual void sendPacket(const char pktName, int pktType)* generates a virtual packet and transmits it down to the lower layer.
Inherited and Protected Member Function Summary
*virtual void handleSelfMsg(cMessage *)* implements the necessary timer for application protocol e.g. sensing timers by sending self-messages. Thus, this method responses to handle all timers and initiate action when timeouts.
*virtual void handleLowerMsg(cMessage *)* handles the message that has sent from an lower layer. After processing this messages, it can be forwarded to lower layers by using *sendUp*() function.
*virtual void handleLowerControl(cMessage * msg)* handles the control message that has sent from a lower layer.
*virtual void handleUpperMsg(cMessage * m)* handles the message that has sent from an upper layer. After processing this messages, it can be forwarded to lower layers by using *sendDown*() function.
*virtual void handleUpperControl(cMessage * m)* handles the control message that has sent from a upper layer.

B.3 Parameter Format

B.3.1 *chnPar* XML Format

```
<?xml version="1.0" encoding="UTF-8"?>
<root>
    <channel   cforwarder="1"    <!-- forwarder node id -->
               cPktId="1002"     <!-- packet id -->
               cPkt="P2"         <!-- packet name -->
               cSrc="2"          <!-- initial source of packet -->
               cSeqNo="1"        <!-- sequence number -->
               cParent="-1"      <!-- next (parent) node id, -1 indicates broadcast -->
               cHopCnt = "1"     <!-- hop count -->
               cData="1261"      <!-- data filed -->
               port="7002"       <!-- port id -->
    />
</root>
```

B.3.2 *NbrLst* XML Format

```
<?xml version="1.0" encoding="UTF-8"?>
<root>
    < neighbour  list="2,3,4,5,6"   <!-- neighbour list -->
                 fPkt = "P2"        <!-- forwarded packet -->
                 fwdr = "1"         <!-- forwarder node id -->
    />
</root>
```

Appendix C

The Event-B Model of the RTMCS Case Study

C.1 Abstract specification: basic communication protocol

C.1.1 Context: C0

CONTEXT C0

SETS

PKT

CONSTANTS

ND

$Destination$

$BROADCAST$

$initialSrcAddr$

AXIOMS

axm0_1 : $ND \subseteq \mathbb{N}$ //set of nodes in the network

axm0_2 : $finite(ND)$ //nodes should be finite

axm0_3 : $Destination \subseteq ND$

//a destination node should be a member in set of nodes

axm0_4 : $initialSrcAddr \in PKT \rightarrow ND$ //a packet structure: source of packet

axm0_5 : $BROADCAST = -1$ //broadcast code

END

C.1.2 Machine: M0

MACHINE M0

SEES C0

VARIABLES

$middleware$ //middleware

$destBuff$ //a destination buffer

$xmittedPkts$ //transmitted packets

$finalDestAddr$ //final destination address

$lostPkts$ //lost packet

INVARIANTS

int0_1 : $xmittedPkts \subseteq PKT$

inv0_2 : $middleware \subseteq PKT$

inv0_3 : $destBuff \in ND \rightarrow \mathbb{P}(PKT)$

inv0_4 : $finalDestAddr \in xmittedPkts \nrightarrow ND$

inv0_5 : $lostPkts \subseteq PKT$

EVENTS

Initialisation

begin

int0_1 : $xmittedPkts := \varnothing$

int0_2 : $middleware := \varnothing$

int0_3 : $destBuff := ND \times \{\varnothing\}$

int0_4 : $finalDestAddr := \varnothing$

int0_5 : $lostPkts := \varnothing$

end

Event $create\_pkt \mathrel{\widehat{=}}$

//nodes create a packet

any

s

$fDes$

pkt

where

g1 : $pkt \in PKT \setminus (xmittedPkts \cup middleware)$

g2 : $s = initialSrcAddr(pkt)$

g3 : $fDes \in ND$

g4 : $pkt \mapsto fDes \notin finalDestAddr$

g5 : $fDes \neq s$

then

a1 : $xmittedPkts := xmittedPkts \cup \{pkt\}$
a2 : $finalDestAddr := finalDestAddr \cup \{pkt \mapsto fDes\}$
a3 : $middleware := middleware \cup \{pkt\}$

end

Event *dest_recv_pkt* $\widehat{=}$

//destinations receives a packet

any

s
des
pkt

where

g1 : $pkt \in PKT$
g2 : $pkt \in dom(finalDestAddr) \wedge des = finalDestAddr(pkt)$
g3 : $des \in dom(destBuff)$
g4 : $pkt \in middleware \setminus destBuff(des)$
g5 : $s = initialSrcAddr(pkt)$

then

a1 : $destBuff(des) := destBuff(des) \cup \{pkt\}$

end

Event *final_tx_pkt* $\widehat{=}$

//the transmitted packet is removed from middleware indicating completion of transmission mechanism

any

pkt
sk

where

g1 : $pkt \in PKT$
g2 : $pkt \in middleware$
g3 : $pkt \in dom(finalDestAddr) \wedge sk = finalDestAddr(pkt)$
g4 : $pkt \in destBuff(sk)$

then

a1 : $middleware := middleware \setminus \{pkt\}$

end

Event *net_losing* $\widehat{=}$

//network loses packet

any

pkt

where

g1 : $pkt \in middleware \setminus lostPkts$

then

a1 : $middleware := middleware \setminus \{pkt\}$

a2 : $lostPkts := lostPkts \cup \{pkt\}$

end

END

C.2 First refinement: neighbour nodes

C.2.1 Context: C1

CONTEXT C1

EXTENDS C0

SETS

$TYPE$

CONSTANTS

$DATA$

$CONTROL$

$type$

$FAILED\_XMIT$

AXIOMS

axm1_1 : $DATA \in TYPE$

axm1_2 : $CONTROL \subseteq TYPE$

axm1_2 : $partition(TYPE, CONTROL, \{DATA\})$

//two packet types: control and data packets

axm1_3 : $type \in PKT \rightarrow TYPE$ //type of packets

axm1_4 : $FAILED\_XMIT \in \mathbb{Z}$ //constant for failed transmission

END

C.2.2 Machine: M1

MACHINE M1

REFINES M0

SEES C1

VARIABLES

$destBuff$ //nodes' receiving buffer

$finalDestAddr$ //final destination address

$ndBuff$ //nodes' waiting buffer

$WiMedium$ //wireless medium

$sensingNDs$ //sensing nodes

$sensedPkts$ //sensed packet

$floodedPkts$ //flooded packets

$floodTbl$ //historical received packets used for detect duplicated packets

$ctlNeighbours$ //neighbour lists

$recLostPkts$ //recorded lost packet packet at each node

$floodFlg$ //flood flag

INVARIANTS

inv1_1 : $ndBuff \in ND \leftrightarrow PKT$

inv1_2 : $WiMedium \in ND \leftrightarrow PKT$

inv1_3 : $sensingNDs \subseteq ND \setminus Destination$

inv1_4 : $sensedPkts \subseteq PKT$

inv1_6 : $floodedPkts \subseteq PKT$

inv1_7 : $floodTbl \in ND \rightarrow \mathbb{P}(PKT)$

inv1_8 : $ctlNeighbours \in PKT \leftrightarrow (ND \cup \{FAILED\_XMIT\})$

inv1_12 : $floodFlg \in ND \rightarrow BOOL$

inv1_20 : $recLostPkts \in ND \leftrightarrow PKT$

glu1_1 : $ran(ndBuff \cup WiMedium) = middleware$

glu1_2 : $sensedPkts \cup floodedPkts = xmittedPkts$

glu_1_3 : $ran(recLostPkts) = lostPkts$

saf1_1 : $ndBuff \cap WiMedium = \varnothing$

saf1_2 : $(ndBuff \cap WiMedium) \cap recLostPkts = \varnothing$

EVENTS

Initialisation

begin

int0_3 : $destBuff := ND \times \{\varnothing\}$

int0_4 : $finalDestAddr := \varnothing$

int1_1 : $ndBuff := \varnothing$

int1_2 : $WiMedium := \varnothing$

int1_3 : $sensingNDs := \varnothing$

int1_4 : $sensedPkts := \varnothing$

int1_6 : $floodedPkts := \varnothing$

int1_7 : $floodTbl := ND \times \{\varnothing\}$

int1_8 : $ctlNeighbours := \varnothing$

int1_12 : $floodFlg := ND \times \{FALSE\}$

inv1_20 : $recLostPkts := \varnothing$

end

Event *create_dataPkt* $\widehat{=}$

//creating a data packet

refines *create_pkt*

any

s

pkt

$fDes$

where

g1 : $pkt \in PKT \wedge s \in ND \setminus Destination$

g2 : $pkt \notin ran(ndBuff \cup WiMedium)$

g3 : $s \notin sensingNDs \wedge pkt \notin (sensedPkts \cup floodedPkts)$

//has not been created yet

g4 : $s = initialSrcAddr(pkt)$

g5 : $s \in dom(floodTbl) \wedge pkt \notin floodTbl(s)$

g6 : $type(pkt) = DATA$

g7 : $fDes \in Destination$

g8 : $pkt \mapsto fDes \notin finalDestAddr$

g9 : $fDes \neq s$

then

a1 : $sensingNDs := sensingNDs \cup \{s\}$

a2 : $sensedPkts := sensedPkts \cup \{pkt\}$

a3 : $ndBuff := ndBuff \cup \{s \mapsto pkt\}$

a4 : $floodTbl(s) := floodTbl(s) \cup \{pkt\}$

a5 : $finalDestAddr := finalDestAddr \cup \{pkt \mapsto fDes\}$

end

Event *clear_sensingNDs* $\widehat{=}$

//clearing a sensed node

any

x

where

g1 : $sensingNDs = \{x\}$

then

a1 : $sensingNDs := \varnothing$

end

Event *create_controlPkt* $\widehat{=}$

//creating a control packet

refines *create_pkt*

any

s

pkt

$fDes$

where

g1 : $pkt \in PKT$
g2 : $pkt \notin ran(ndBuff \cup WiMedium)$
g3 : $pkt \notin (sensedPkts \cup floodedPkts)$ //has not been created yet
g4 : $s = initialSrcAddr(pkt)$
g5 : $s \in dom(floodTbl) \wedge pkt \notin floodTbl(s)$
g6 : $type(pkt) \in CONTROL$
g7 : $floodFlg(s) = TRUE$
g8 : $fDes \in ND$
g9 : $pkt \mapsto fDes \notin finalDestAddr$
g10 : $fDes \neq s$

then

a1 : $floodedPkts := floodedPkts \cup \{pkt\}$
a2 : $ndBuff := ndBuff \cup \{s \mapsto pkt\}$
a3 : $floodTbl(s) := floodTbl(s) \cup \{pkt\}$
a4 : $finalDestAddr := finalDestAddr \cup \{pkt \mapsto fDes\}$

end

Event *start_tx* $\widehat{=}$

//starting transmiting a packet to wireless medium

any

x
pkt

where

g1 : $x \mapsto pkt \in ndBuff$ //packet in waiting buffer
g2 : $x \mapsto pkt \notin WiMedium$ //but not has been transmitted to wireless medium

then

a1 : $ndBuff := ndBuff \setminus \{x \mapsto pkt\}$
a2 : $WiMedium := WiMedium \cup \{x \mapsto pkt\}$

end

Event *indicate_neighbours* $\widehat{=}$

//channel indicates the neighbour of a forwarder

any

pkt
$nbrs$
f

where

g1 : $pkt \in ran(WiMedium)$
g2 : $f \mapsto pkt \in WiMedium$
g3 : $nbrs \in \{n | n \in (\mathbb{P}(ND)) \cup \{\{FAILED\_XMIT\}\}\}$
g4 : $nbrs \neq \varnothing$

g5 : $pkt \notin dom(ctlNeighbours)$

then

a1 : $ctlNeighbours := ctlNeighbours \cup (\{pkt\} \times nbrs)$

end

Event *receive_pkt* $\widehat{=}$

//each neighbour node receives a transmitted packet

any

s

nb

pkt

where

g1 : $pkt \in ran(WiMedium)$

g2 : $pkt \mapsto nb \in ctlNeighbours$

g3 : $nb \mapsto pkt \notin ndBuff$

g4 : $nb \mapsto pkt \notin WiMedium$

g5 : $s = initialSrcAddr(pkt) \wedge s \neq nb$

g6 : $pkt \in dom(finalDestAddr) \wedge nb \neq finalDestAddr(pkt)$

g7 : $nb \in dom(floodTbl) \wedge pkt \notin floodTbl(nb)$ //is not duplicted packet

then

a1 : $ndBuff := ndBuff \cup \{nb \mapsto pkt\}$

a2 : $ctlNeighbours := ctlNeighbours \setminus \{pkt \mapsto nb\}$

a3 : $floodTbl(nb) := floodTbl(nb) \cup \{pkt\}$

end

Event *receive_dup_pkt* $\widehat{=}$

//each neighbour node discards a duplicated packet

any

nb

pkt

where

g1 : $pkt \in dom(ctlNeighbours)$

g2 : $pkt \mapsto nb \in ctlNeighbours$

g3 : $nb \in dom(floodTbl) \wedge pkt \in floodTbl(nb)$ //is a duplicted packet

then

a1 : $ctlNeighbours := ctlNeighbours \setminus \{pkt \mapsto nb\}$

end

Event *dest_recv_pkt* $\widehat{=}$

//destination receives a packet

refines *dest_recv_pkt*

any

s

des

pkt

where

g1 : $pkt \in PKT$

g2 : $pkt \in dom(finalDestAddr) \wedge des = finalDestAddr(pkt)$

g3 : $des \in dom(destBuff)$

g4 : $pkt \in ran(WiMedium) \setminus destBuff(des)$

g5 : $s = initialSrcAddr(pkt)$

g6 : $\{pkt\} \lhd ctlNeighbours \neq \varnothing$

g7 : $ran(\{pkt\} \lhd ctlNeighbours) \neq \{FAILED\_XMIT\}$

g8 : $des \in (ran(\{pkt\} \lhd ctlNeighbours))$ //neighbour who is a destination

then

a1 : $destBuff(des) := destBuff(des) \cup \{pkt\}$

a2 : $ctlNeighbours := ctlNeighbours \setminus \{pkt \mapsto des\}$

a3 : $floodTbl(des) := floodTbl(des) \cup \{pkt\}$

end

Event *finish_tx_pkt* $\widehat{=}$

//finishing packet transmission

any

pkt

f

where

g1 : $\{pkt\} \lhd ctlNeighbours = \varnothing$

g2 : $pkt \in ran(WiMedium) \wedge pkt \in ran(ndBuff)$

//there are some packets being forwarded

g3 : $f \in ND \wedge f \mapsto pkt \in WiMedium$

then

a1 : $WiMedium := WiMedium \setminus \{f \mapsto pkt\}$

end

Event *final_tx_pkt* $\widehat{=}$

//the transmitted packet is removed from middleware indicating completion of transmission mechanism

refines *final_tx_pkt*

any

pkt

sk

where

g1 : $pkt \in PKT$

g2 : $pkt \in ran(WiMedium) \wedge pkt \notin ran(ndBuff)$

//in WiMedium and not transited in another nodes

g3 : $pkt \in dom(finalDestAddr) \wedge sk = finalDestAddr(pkt)$

g4 : $sk \in ND$
g5 : $pkt \in destBuff(sk)$
g6 : $\{pkt\} \lhd ctlNeighbours = \varnothing$ //all neighbours receive transmitted packet pkt

then

a1 : $WiMedium := WiMedium \rhd \{pkt\}$

end

Event *lose_pkt* $\widehat{=}$

//each node records a lost packet

refines *net_losing*

any

f
pkt

where

g1 : $pkt \in ran(WiMedium) \setminus ran(recLostPkts)$
g2 : $pkt \notin ran(ndBuff)$
g3 : $f \mapsto pkt \in WiMedium$
g4 : $f \mapsto pkt \notin recLostPkts$
g5 : $ran(\{pkt\} \lhd ctlNeighbours) = \{FAILED\_XMIT\}$

then

a1 : $WiMedium := WiMedium \rhd \{pkt\}$
a2 : $recLostPkts := recLostPkts \cup \{f \mapsto pkt\}$
a3 : $ctlNeighbours := ctlNeighbours \setminus \{pkt \mapsto FAILED\_XMIT\}$

end

END

C.3 Second refinement: sensing and actuating mechanism

C.3.1 Context: C2

CONTEXT C2

EXTENDS C1

CONSTANTS

$Sink$
$Actuators$
$safetyThreshold$
$randomFn$
$minR$
$maxR$

AXIOMS

axm2_1 : $Sink \in Destination$

axm2_2 : $Actuators \subseteq Destination$

axm2_3 : $partition(Destination, \{Sink\}, Actuators)$

//destination is partitioned into two types: a sink and actuators

axm2_4 : $safetyThreshold \in \mathbb{Z}$ //constants for safety threshold

axm2_5 : $randomFn = minR \mathinner{..} maxR$
//basic random function for environment

END

C.3.2 Machine: M2

MACHINE M2

REFINES M1

SEES C2

VARIABLES

$destBuff$ //nodes' receiving buffer

$finalDestAddr$ //final destination address

$ndBuff$ //nodes' waiting buffer

$WiMedium$ //wireless medium

$sensingNDs$ //sensing nodes

$sensedPkts$ //sensed packets

$floodedPkts$ //flooded packets

$floodTbl$ //historical received packets used for detect duplicated packets

$ctlNeighbours$ //neighbour lists

$recLostPkts$ //recorded lost packet packet at each node

$ctlSensedFlg$ //sensing period

$floodFlg$ //flood flag

$envSensedFlg$ //sensing period

$envData$ //environmental/conditional data

$senseBuff$ //sense ndBuff

$recvedData$ //received data at destination

$emergencyAlert$ //actuating status

$pktData$ //data attached in transmitted packet

$vPktData$ //virtual packet at channel

INVARIANTS

inv2_1 : $envSensedFlg \in ND \setminus Destination \rightarrow BOOL$

inv2_2 : $envData \in ND \setminus Destination \rightarrow \mathbb{Z}$

inv2_3 : $ctlSensedFlg \in ND \setminus Destination \rightarrow BOOL$

inv2_4 : $senseBuff \in ND \setminus Destination \rightarrow \mathbb{P}(\mathbb{Z})$

inv2_5 : $recvedData \in Destination \rightarrow \mathbb{P}(\mathbb{Z})$

inv2_6 : $emergencyAlert \in Actuators \rightarrow BOOL$

inv2_7 : $pktData \in PKT \nrightarrow \mathbb{Z}$

inv2_8 : $vPktData \in PKT \nrightarrow \mathbb{Z}$

EVENTS

Initialisation

extended

begin

int0_3 : $destBuff := ND \times \{\varnothing\}$

int0_4 : $finalDestAddr := \varnothing$

int1_1 : $ndBuff := \varnothing$

int1_2 : $WiMedium := \varnothing$

int1_3 : $sensingNDs := \varnothing$

int1_4 : $sensedPkts := \varnothing$

int1_6 : $floodedPkts := \varnothing$

int1_7 : $floodTbl := ND \times \{\varnothing\}$

int1_8 : $ctlNeighbours := \varnothing$

int1_12 : $floodFlg := ND \times \{FALSE\}$

inv1_20 : $recLostPkts := \varnothing$

int2_1 : $envSensedFlg := (ND \setminus Destination) \times \{FALSE\}$

int2_2 : $envData := (ND \setminus Destination) \times \{0\}$

int2_3 : $ctlSensedFlg := (ND \setminus Destination) \times \{FALSE\}$

int2_4 : $senseBuff := (ND \setminus Destination) \times \{\varnothing\}$

int2_5 : $recvedData := Destination \times \{\varnothing\}$

int2_6 : $emergencyAlert := Actuators \times \{FALSE\}$

int2_7 : $pktData := \varnothing$

int2_8 : $vPktData := \varnothing$

end

Event *start_sensing* $\widehat{=}$

//activating each node to sense data

any

x

sd

where

g1 : $x \in ND \setminus Destination$

g2 : $envSensedFlg(x) = FALSE$

g3 : $sd \in randomFn$

then

a1 : $envSensedFlg(x) := TRUE$

a2 : $envData(x) := sd$

end

Event *sensing* $\widehat{=}$

//each node sense a data from environment

any

x

sf

sd

where

g1 : $x \in ND \setminus Destination$

g2 : $sf = envSensedFlg(x)$

g3 : $envSensedFlg(x) = TRUE$

g4 : $ctlSensedFlg(x) = FALSE$

g5 : $sd = envData(x)$

then

a1 : $ctlSensedFlg(x) := sf$

a2 : $senseBuff(x) := senseBuff(x) \cup \{sd\}$

end

Event *finish_sensing* $\widehat{=}$

//finishing sensing data process

extends *clear_sensingNDs*

any

x

where

g1 : $sensingNDs = \{x\}$

g2 : $x \in dom(envSensedFlg)$

g3 : $envSensedFlg(x) = TRUE$

g4 : $ctlSensedFlg(x) = TRUE$

then

a1 : $sensingNDs := \varnothing$

a2 : $ctlSensedFlg(x) := FALSE$

a3 : $envSensedFlg(x) := FALSE$

end

Event *create_dataPkt* $\widehat{=}$

//creating a data packet

extends *create_dataPkt*

any

s

pkt

$fDes$

$data$

where

g1 : $pkt \in PKT \wedge s \in ND \setminus Destination$

g2 : $pkt \notin ran(ndBuff \cup WiMedium)$

g3 : $s \notin sensingNDs \wedge pkt \notin (sensedPkts \cup floodedPkts)$

//has not been created yet

g4 : $s = initialSrcAddr(pkt)$

g5 : $s \in dom(floodTbl) \wedge pkt \notin floodTbl(s)$

g6 : $type(pkt) = DATA$

g7 : $fDes \in Destination$

g8 : $pkt \mapsto fDes \notin finalDestAddr$

g9 : $fDes \neq s$

g10 : $ctlSensedFlg(s) = TRUE$

g11 : $s \in dom(senseBuff) \wedge senseBuff \neq \varnothing \wedge data \in senseBuff(s)$

g12 : $pkt \notin dom(pktData)$

then

a1 : $sensingNDs := sensingNDs \cup \{s\}$

a2 : $sensedPkts := sensedPkts \cup \{pkt\}$

a3 : $ndBuff := ndBuff \cup \{s \mapsto pkt\}$

a4 : $floodTbl(s) := floodTbl(s) \cup \{pkt\}$

a5 : $finalDestAddr := finalDestAddr \cup \{pkt \mapsto fDes\}$

a6 : $senseBuff(s) := senseBuff(s) \setminus \{data\}$

a7 : $pktData := pktData \cup \{pkt \mapsto data\}$

end

Event *create_controlPkt* $\widehat{=}$

//creating a control packet

extends *create_controlPkt*

any

s

pkt

$fDes$

where

g1 : $pkt \in PKT$

g2 : $pkt \notin ran(ndBuff \cup WiMedium)$

g3 : $pkt \notin (sensedPkts \cup floodedPkts)$ //has not been created yet

g4 : $s = initialSrcAddr(pkt)$

g5 : $s \in dom(floodTbl) \wedge pkt \notin floodTbl(s)$

g6 : $type(pkt) \in CONTROL$

g7 : $floodFlg(s) = TRUE$
g8 : $fDes \in ND$
g9 : $pkt \mapsto fDes \notin finalDestAddr$
g10 : $fDes \neq s$
g11 : $pkt \notin dom(pktData)$

then

a1 : $floodedPkts := floodedPkts \cup \{pkt\}$
a2 : $ndBuff := ndBuff \cup \{s \mapsto pkt\}$
a3 : $floodTbl(s) := floodTbl(s) \cup \{pkt\}$
a4 : $finalDestAddr := finalDestAddr \cup \{pkt \mapsto fDes\}$
a5 : $pktData := pktData \cup \{pkt \mapsto 0\}$

end

Event *start_tx* $\widehat{=}$

//starting transmiting a packet to wireless medium

extends *start_tx*

any

x
pkt

where

g1 : $x \mapsto pkt \in ndBuff$ //packet in waiting buffer
g2 : $x \mapsto pkt \notin WiMedium$ //but not has been transmitted to wireless medium

then

a1 : $ndBuff := ndBuff \setminus \{x \mapsto pkt\}$
a2 : $WiMedium := WiMedium \cup \{x \mapsto pkt\}$

end

Event *assign_vPktData* $\widehat{=}$

//assigning virtual data packet

any

pkt
$data$

where

g1 : $pkt \notin dom(vPktData)$
g2 : $pkt \in dom(pktData) \wedge data = pktData(pkt)$

then

a1 : $vPktData := vPktData \cup \{pkt \mapsto data\}$
a2 : $pktData := \{pkt\} \lhd pktData$

end

Event *indicate_neighbours* $\widehat{=}$

//channel indicates the neighbour of a forwarder

extends *indicate_neighbours*

any

pkt
$nbrs$
f
$data$

where

g1 : $pkt \in ran(WiMedium)$
g2 : $f \mapsto pkt \in WiMedium$
g3 : $nbrs \in \{n | n \in (\mathbb{P}(ND)) \cup \{\{FAILED\_XMIT\}\}\}$
g4 : $nbrs \neq \varnothing$
g5 : $pkt \notin dom(ctlNeighbours)$
g6 : $pkt \in dom(vPktData) \wedge data = vPktData(pkt)$
g7 : $pkt \notin dom(pktData)$

then

a1 : $ctlNeighbours := ctlNeighbours \cup (\{pkt\} \times nbrs)$
a2 : $vPktData := \{pkt\} \lhd vPktData$
a3 : $pktData := pktData \cup \{pkt \mapsto data\}$

end

Event *receive_pkt* $\widehat{=}$

//each neighbour node receives a transmitted packet

extends *receive_pkt*

any

s
nb
pkt

where

g1 : $pkt \in ran(WiMedium)$
g2 : $pkt \mapsto nb \in ctlNeighbours$
g3 : $nb \mapsto pkt \notin ndBuff$
g4 : $nb \mapsto pkt \notin WiMedium$
g5 : $s = initialSrcAddr(pkt) \wedge s \neq nb$
g6 : $pkt \in dom(finalDestAddr) \wedge nb \neq finalDestAddr(pkt)$
g7 : $nb \in dom(floodTbl) \wedge pkt \notin floodTbl(nb)$ //is not duplicted packet

then

a1 : $ndBuff := ndBuff \cup \{nb \mapsto pkt\}$
a2 : $ctlNeighbours := ctlNeighbours \setminus \{pkt \mapsto nb\}$
a3 : $floodTbl(nb) := floodTbl(nb) \cup \{pkt\}$

end

Event *receive_dup_pkt* $\widehat{=}$

//each neighbour node discards a duplicated packet

extends *receive_dup_pkt*

any

nb

pkt

where

g1 : $pkt \in dom(ctlNeighbours)$

g2 : $pkt \mapsto nb \in ctlNeighbours$

g3 : $nb \in dom(floodTbl) \land pkt \in floodTbl(nb)$ //is a duplicted packet

then

a1 : $ctlNeighbours := ctlNeighbours \setminus \{pkt \mapsto nb\}$

end

Event *dest_recv_dataPkt* $\widehat{=}$

//destination receives a data packet

extends *dest_recv_pkt*

any

s

des

pkt

$data$

where

g1 : $pkt \in PKT$

g2 : $pkt \in dom(finalDestAddr) \land des = finalDestAddr(pkt)$

g3 : $des \in dom(destBuff)$

g4 : $pkt \in ran(WiMedium) \setminus destBuff(des)$

g5 : $s = initialSrcAddr(pkt)$

g6 : $\{pkt\} \lhd ctlNeighbours \neq \varnothing$

g7 : $ran(\{pkt\} \lhd ctlNeighbours) \neq \{FAILED\_XMIT\}$

g8 : $des \in (ran(\{pkt\} \lhd ctlNeighbours))$ //neighbour who is a destination

g9 : $des \in dom(recvedData)$

g10 : $pkt \in dom(pktData) \land data = pktData(pkt)$

g11 : $type(pkt) = DATA$

then

a1 : $destBuff(des) := destBuff(des) \cup \{pkt\}$

a2 : $ctlNeighbours := ctlNeighbours \setminus \{pkt \mapsto des\}$

a3 : $floodTbl(des) := floodTbl(des) \cup \{pkt\}$

a4 : $recvedData(des) := recvedData(des) \cup \{data\}$

end

Event *dest_recv_controlPkt* $\widehat{=}$

//destination receives a control packet

extends *dest_recv_pkt*

any

s

des

pkt

where

g1 : $pkt \in PKT$

g2 : $pkt \in dom(finalDestAddr) \wedge des = finalDestAddr(pkt)$

g3 : $des \in dom(destBuff)$

g4 : $pkt \in ran(WiMedium) \setminus destBuff(des)$

g5 : $s = initialSrcAddr(pkt)$

g6 : $\{pkt\} \lhd ctlNeighbours \neq \varnothing$

g7 : $ran(\{pkt\} \lhd ctlNeighbours) \neq \{FAILED\_XMIT\}$

g8 : $des \in (ran(\{pkt\} \lhd ctlNeighbours))$ //neighbour who is a destination

g9 : $type(pkt) \in CONTROL$

then

a1 : $destBuff(des) := destBuff(des) \cup \{pkt\}$

a2 : $ctlNeighbours := ctlNeighbours \setminus \{pkt \mapsto des\}$

a3 : $floodTbl(des) := floodTbl(des) \cup \{pkt\}$

end

Event *actuating* $\widehat{=}$

//alerting emergency: sensed data is above safety threshold

any

act

$data$

where

g1 : $act \in Actuators$

g2 : $act \in dom(recvedData)$

g3 : $data \in recvedData(act)$

g4 : $data \geq safetyThreshold$

g5 : $emergencyAlert(act) = FALSE$

then

a1 : $recvedData(act) := recvedData(act) \setminus \{data\}$

a2 : $emergencyAlert(act) := TRUE$

end

Event *no_actuating* $\widehat{=}$

//not alerting emergency sensed data is below safety threshold

any

act

$data$

where

g1 : $act \in Actuators$

g2 : $act \in dom(recvedData)$

g3 : $data \in recvedData(act)$
g4 : $data < safetyThreshold$
then

a1 : $recvedData(act) := recvedData(act) \setminus \{data\}$
end

Event *reset_actuatingStatus* $\widehat{=}$
//resetting emergency alert status
any

act
where

g1 : $act \in dom(emergencyAlert)$
g2 : $emergencyAlert(act) = TRUE$
then

a1 : $emergencyAlert(act) := FALSE$
end

Event *finish_tx_pkt* $\widehat{=}$
//finishing packet transmission
extends *finish_tx_pkt*
any

pkt
f
where

g1 : $\{pkt\} \lhd ctlNeighbours = \varnothing$
g2 : $pkt \in ran(WiMedium) \wedge pkt \in ran(ndBuff)$
//there are some packets being forwarded
g3 : $f \in ND \wedge f \mapsto pkt \in WiMedium$
then

a1 : $WiMedium := WiMedium \setminus \{f \mapsto pkt\}$
end

Event *final_tx_pkt* $\widehat{=}$
//the transmitted packet is removed from middleware indicating completion of transmission mechanism
extends *final_tx_pkt*
any

pkt
sk
where

g1 : $pkt \in PKT$
g2 : $pkt \in ran(WiMedium) \wedge pkt \notin ran(ndBuff)$
//in WiMedium and not transited in another nodes

g3 : $pkt \in dom(finalDestAddr) \wedge sk = finalDestAddr(pkt)$
g4 : $sk \in ND$
g5 : $pkt \in destBuff(sk)$
g6 : $\{pkt\} \lhd ctlNeighbours = \varnothing$ //all neighbours receive transmitted packet pkt

then

a1 : $WiMedium := WiMedium \rhd \{pkt\}$

end

Event *lose_pkt* $\widehat{=}$

//each node records a lost packet

extends *lose_pkt*

any

f
pkt

where

g1 : $pkt \in ran(WiMedium) \setminus ran(recLostPkts)$
g2 : $pkt \notin ran(ndBuff)$
g3 : $f \mapsto pkt \in WiMedium$
g4 : $f \mapsto pkt \notin recLostPkts$
g5 : $ran(\{pkt\} \lhd ctlNeighbours) = \{FAILED\_XMIT\}$

then

a1 : $WiMedium := WiMedium \rhd \{pkt\}$
a2 : $recLostPkts := recLostPkts \cup \{f \mapsto pkt\}$
a3 : $ctlNeighbours := ctlNeighbours \setminus \{pkt \mapsto FAILED\_XMIT\}$

end

END

C.4 Third refinement: network topology, physical environment and channel mechanism

C.4.1 Context: C3

CONTEXT C3

EXTENDS C2

CONSTANTS

WSN
$wsnTopology$

AXIOMS

axm3_1 : $WSN = ND \leftrightarrow ND$ //wireless sensor network structure

axm3_2 : $wsnTopology \in WSN$ //wireless sensor network topology

END

C.4.2 Machine: M3

MACHINE M3

REFINES M2

SEES C3

VARIABLES

$destBuff$ //(ctl) nodes' receiving buffer

$finalDestAddr$ //(ctl) final destination address

$ndBuff$ //(ctl) nodes' waiting buffer

$sensingNDs$ //(ctl) sensing nodes

$sensedPkts$ //(ctl)sensed packets

$floodedPkts$ //(ctl) flooded packets

$floodTbl$ //(ctl) historical received packets used for detect duplicated packets

$ctlNeighbours$ //(ctl) neighbour lists

$recLostPkts$ //(ctl) record recLostPkts packet at each node

$floodFlg$ //(ctl) flood flag

$envSensedFlg$ //(env) sensing period

$envData$ //(env) environmental/conditional data

$ctlSensedFlg$ //(ctl)sensing period

$senseBuff$ //(ctl) sense buffer

$recvedData$ //(ctl) received data at destination

$emergencyAlert$ //(ctl) actuating status

$pktData$ //(ctl) data attached in transmitted packet

$sentDown$ //(ctl) transmitted packet to channel

$sentUp$ //(ctl) transmitted packet up to channel

$channel$ //(ctl) channel

$envNeighbours$ //(env) neighbour list

$wsnLinks$ //(env) dynamic WSN links

$crashedLinks$ //(env) crashed WSN Links

$dataSeqNo$ //(ctl) local nodes' current data sequence number at application layer

$floodSeqNo$ //(ctl) local nodes' current broadcast sequence number at application layer

$nbHops$ //(ctl) neighbour's hop count

$pktSrc$ //(ctl) src address attached in transmitted packet

$pktSeqNo$ //(ctl) sequence number attached in transmitted packet

$pktFwdr$ //(ctl) fowarder attached in transmitted packet

$pktNbHops$ //(ctl) hop count attached in transmitted packet

$vPktSeqNo$ //(env) virtual packet at channel

$vPktSrc$ //(env) virtual packet at channel

$vPktFwdr$ //(env) virtual packet at channel

$vPktData$ //(env) virtual packet at channel

$vPktNbHops$ //(env) virtual packet at chennel

INVARIANTS

inv3_1 : $sentUp \in ND \leftrightarrow PKT$

inv3_2 : $sentDown \in ND \leftrightarrow PKT$

inv3_3 : $channel \in ND \leftrightarrow PKT$

inv3_4 : $envNeighbours \in ran(channel) \leftrightarrow (ND \cup \{FAILED\_XMIT\})$

inv3_5 : $wsnLinks \in WSN$

inv3_6 : $crashedLinks \in WSN$

inv3_7 : $dataSeqNo \in ND \rightarrow \mathbb{N}$

inv3_8 : $floodSeqNo \in ND \rightarrow \mathbb{N}$

inv3_9 : $nbHops \in ND \leftrightarrow (PKT \nrightarrow \mathbb{Z})$

inv3_10 : $pktSeqNo \in PKT \nrightarrow \mathbb{N}$

inv3_11 : $pktSrc \in PKT \nrightarrow ND$

inv3_12 : $pktFwdr \in PKT \nrightarrow ND$

inv3_13 : $vPktNbHops \in ran(channel) \nrightarrow \mathbb{Z}$

inv3_14 : $vPktSrc \in ran(channel) \nrightarrow ND$

inv3_15 : $pktNbHops \in PKT \nrightarrow \mathbb{Z}$

inv3_16 : $vPktSeqNo \in ran(channel) \nrightarrow \mathbb{N}$

inv3_17 : $vPktFwdr \in ran(channel) \nrightarrow ND$

saf3_1 : $partition(WiMedium, sentDown, sentUp)$

EVENTS

Initialisation

begin

int0_3 : $destBuff := ND \times \{\varnothing\}$

int0_4 : $finalDestAddr := \varnothing$

int1_1 : $ndBuff := \varnothing$

int1_3 : $sensingNDs := \varnothing$

int1_4 : $sensedPkts := \varnothing$

int1_6 : $floodedPkts := \varnothing$

int1_7 : $floodTbl := ND \times \{\varnothing\}$

int1_8 : $ctlNeighbours := \varnothing$

int1_12 : $floodFlg := ND \times \{FALSE\}$
inv1_20 : $recLostPkts := \varnothing$
int2_1 : $envSensedFlg := (ND \setminus Destination) \times \{FALSE\}$
int2_2 : $envData := (ND \setminus Destination) \times \{0\}$
int2_3 : $ctlSensedFlg := (ND \setminus Destination) \times \{FALSE\}$
int2_4 : $senseBuff := (ND \setminus Destination) \times \{\varnothing\}$
int2_5 : $recvedData := Destination \times \{\varnothing\}$
int2_6 : $emergencyAlert := Actuators \times \{FALSE\}$
int2_7 : $pktData := \varnothing$
int3_1 : $sentUp := \varnothing$
int3_2 : $sentDown := \varnothing$
int3_3 : $channel := \varnothing$
int3_4 : $envNeighbours := \varnothing$
int3_5 : $wsnLinks := \varnothing$
int3_6 : $crashedLinks := \varnothing$
int3_7 : $dataSeqNo := ND \times \{0\}$
int3_8 : $floodSeqNo := ND \times \{0\}$
int3_9 : $nbHops := \varnothing$
int3_10 : $pktSeqNo := \varnothing$
int3_11 : $pktSrc := \varnothing$
int3_12 : $pktFwdr := \varnothing$
int3_13 : $pktNbHops := \varnothing$
int3_14 : $vPktSeqNo := \varnothing$
int3_15 : $vPktSrc := \varnothing$
int3_16 : $vPktFwdr := \varnothing$
int2_8 : $vPktData := \varnothing$
int3_18 : $vPktNbHops := \varnothing$

end

Event *add_link* $\widehat{=}$

//(env) channel sets a topology

any

x
y

where

g1 : $x \in ND \wedge y \in ND$
g2 : $x \mapsto y \notin wsnLinks$
g3 : $x \neq y$

then

a1 : $wsnLinks := wsnLinks \cup \{x \mapsto y\}$

end

Event *disconnect_link* $\widehat{=}$

//(env) disconnecting a link

any

x

y

where

g1 : $x \in ND \wedge y \in ND$

g2 : $x \mapsto y \in wsnLinks$

g3 : $x \mapsto y \notin crashedLinks$

then

a1 : $crashedLinks := crashedLinks \cup \{x \mapsto y\}$

a2 : $wsnLinks := wsnLinks \setminus \{x \mapsto y\}$

end

Event *recover_link* $\widehat{=}$

//(env) recovering a link

any

x

y

where

g1 : $x \in ND \wedge y \in ND$

g2 : $x \mapsto y \in crashedLinks$

g3 : $x \mapsto y \notin wsnLinks$

then

a1 : $wsnLinks := wsnLinks \cup \{x \mapsto y\}$

a2 : $crashedLinks := crashedLinks \setminus \{x \mapsto y\}$

end

Event *start_sensing* $\widehat{=}$

//(env) activating each node to sense data

extends *start_sensing*

any

x

sd

where

g1 : $x \in ND \setminus Destination$

g2 : $envSensedFlg(x) = FALSE$

g3 : $sd \in randomFn$

then

a1 : $envSensedFlg(x) := TRUE$

a2 : $envData(x) := sd$

end

Event *sensing* $\widehat{=}$

//(shd) each node sense a data from environment

extends *sensing*

any

x

sf

sd

where

g1 : $x \in ND \setminus Destination$

g2 : $sf = envSensedFlg(x)$

g3 : $envSensedFlg(x) = TRUE$

g4 : $ctlSensedFlg(x) = FALSE$

g5 : $sd = envData(x)$

then

a1 : $ctlSensedFlg(x) := sf$

a2 : $senseBuff(x) := senseBuff(x) \cup \{sd\}$

end

Event *finish_sensing* $\widehat{=}$

//(shd) finishing sensing data process

refines *finish_sensing*

any

x

where

g1 : $sensingNDs = \{x\}$

g2 : $x \in dom(envSensedFlg)$

g3 : $envSensedFlg(x) = TRUE$

g4 : $ctlSensedFlg(x) = TRUE$

g5 : $ndBuff = \varnothing \wedge (sentDown \cup sentUp) = \varnothing$

then

a1 : $sensingNDs := \varnothing$

a2 : $ctlSensedFlg(x) := FALSE$

a3 : $envSensedFlg(x) := FALSE$

@a3 sensingNDs := $\varnothing$

end

Event *create_dataPkt* $\widehat{=}$

//(ctl) creating a data packet

refines *create_dataPkt*

any

s

pkt

$fDes$
$data$
sno
nbh

where

g1 : $pkt \in PKT \wedge s \in ND \setminus Destination$
g2 : $pkt \notin ran(ndBuff \cup sentDown \cup sentUp)$
g3 : $s \notin sensingNDs \wedge pkt \notin (sensedPkts \cup floodedPkts)$
//has not been created yet
g4 : $s = initialSrcAddr(pkt)$
g5 : $s \in dom(floodTbl) \wedge pkt \notin floodTbl(s)$ //not is duplicted packet
g6 : $type(pkt) = DATA$
g7 : $fDes \in Destination$
g8 : $pkt \mapsto fDes \notin finalDestAddr$
g9 : $fDes \neq s$
g10 : $ctlSensedFlg(s) = TRUE$
g11 : $s \in dom(senseBuff) \wedge senseBuff \neq \varnothing \wedge data \in senseBuff(s)$
g12 : $pkt \notin dom(pktData)$
g13 : $sno = dataSeqNo(s) + 1 \wedge s \in dom(dataSeqNo)$ //identifing seq number
g14 : $nbh \in \mathbb{Z} \wedge nbh = -1$
g15 : $s \mapsto \{pkt \mapsto nbh\} \notin nbHops$
g16 : $pkt \notin dom(pktSeqNo)$
g17 : $pkt \notin dom(pktSrc)$
g18 : $pkt \notin dom(pktFwdr)$
g19 : $pkt \notin dom(pktNbHops)$

then

a1 : $sensingNDs := sensingNDs \cup \{s\}$
a2 : $sensedPkts := sensedPkts \cup \{pkt\}$
a3 : $ndBuff := ndBuff \cup \{s \mapsto pkt\}$
a4 : $floodTbl(s) := floodTbl(s) \cup \{pkt\}$
a5 : $finalDestAddr := finalDestAddr \cup \{pkt \mapsto fDes\}$
a6 : $senseBuff(s) := senseBuff(s) \setminus \{data\}$
a7 : $pktData := pktData \cup \{pkt \mapsto data\}$
a8 : $dataSeqNo := dataSeqNo \lhd\!\!\!- \{s \mapsto sno\}$
a9 : $nbHops := nbHops \cup \{s \mapsto \{pkt \mapsto nbh\}\}$
a10 : $pktSeqNo := pktSeqNo \lhd\!\!\!- \{pkt \mapsto sno\}$
a11 : $pktSrc := pktSrc \cup \{pkt \mapsto s\}$
a12 : $pktFwdr := pktFwdr \cup \{pkt \mapsto s\}$
a13 : $pktNbHops := pktNbHops \cup \{pkt \mapsto nbh\}$

end

Event *create_controlPkt* $\widehat{=}$

//(ctl) creating a control packet

refines *create_controlPkt*

any

s

pkt

$fDes$

sno

nbh

where

g1 : $pkt \in PKT$

g2 : $pkt \notin ran(ndBuff \cup sentDown \cup sentUp)$

g3 : $pkt \notin (sensedPkts \cup floodedPkts)$

g4 : $s = initialSrcAddr(pkt)$

g5 : $s \in dom(floodTbl) \wedge pkt \notin floodTbl(s)$

g6 : $type(pkt) \in CONTROL$

g7 : $floodFlg(s) = TRUE$

g8 : $fDes \in ND$

g9 : $pkt \mapsto fDes \notin finalDestAddr$

g10 : $fDes \neq s$

g11 : $sno = floodSeqNo(s) + 1 \wedge s \in dom(floodSeqNo)$

g12 : $nbh \in \mathbb{Z} \wedge nbh = -1$

g13 : $s \mapsto \{pkt \mapsto nbh\} \notin nbHops$

g14 : $pkt \notin dom(pktSeqNo)$

g15 : $pkt \notin dom(pktSrc)$

g16 : $pkt \notin dom(pktFwdr)$

g17 : $pkt \notin dom(pktData)$

g18 : $pkt \notin dom(pktNbHops)$

then

a1 : $floodedPkts := floodedPkts \cup \{pkt\}$

a2 : $ndBuff := ndBuff \cup \{s \mapsto pkt\}$

a3 : $floodTbl(s) := floodTbl(s) \cup \{pkt\}$

a4 : $finalDestAddr := finalDestAddr \cup \{pkt \mapsto fDes\}$

a5 : $floodSeqNo := floodSeqNo \Leftarrow \{s \mapsto sno\}$

a6 : $nbHops := nbHops \cup \{s \mapsto \{pkt \mapsto nbh\}\}$

a7 : $pktSeqNo := pktSeqNo \Leftarrow \{pkt \mapsto sno\}$

a8 : $pktFwdr := pktFwdr \cup \{pkt \mapsto s\}$

a9 : $pktSrc := pktSrc \cup \{pkt \mapsto s\}$

a10 : $pktData := pktData \cup \{pkt \mapsto 0\}$

a11 : $pktNbHops := pktNbHops \cup \{pkt \mapsto nbh\}$

end

Event *start_tx* $\widehat{=}$

//(ctl) starting transmiting a packet to a channel

refines *start_tx*

any

x

pkt

nbh

where

g1 : $x \mapsto pkt \in ndBuff$ //packet in waiting buffer

g2 : $x \mapsto pkt \notin sentDown$ //but has not tranmitted down to a channel

g3 : $x \mapsto pkt \notin sentUp$

g4 : $x \mapsto \{pkt \mapsto nbh\} \in nbHops$

g5 : $pkt \in dom(pktFwdr)$

g6 : $pkt \in dom(pktNbHops)$

then

a1 : $ndBuff := ndBuff \setminus \{x \mapsto pkt\}$

a2 : $sentDown := sentDown \cup \{x \mapsto pkt\}$

a3 : $pktFwdr := pktFwdr \lhd\!\!\!- \{pkt \mapsto x\}$

a4 : $pktNbHops := pktNbHops \lhd\!\!\!- \{pkt \mapsto nbh\}$

a5 : $nbHops := nbHops \setminus \{x \mapsto \{pkt \mapsto nbh\}\}$

end

Event *send_down* $\widehat{=}$

//(shd) sending a packet down to a channel

refines *assign_vPktData*

any

cn

pkt

sno

src

$data$

$fwdr$

nbh

where

g1 : $cn \mapsto pkt \in sentDown$

g2 : $cn \mapsto pkt \notin channel$

g3 : $pkt \in dom(pktSeqNo) \wedge sno = pktSeqNo(pkt)$

g4 : $pkt \in dom(pktSrc) \wedge src = pktSrc(pkt)$

g5 : $pkt \in dom(pktFwdr) \wedge fwdr = pktFwdr(pkt)$

g6 : $pkt \in dom(pktData) \wedge data = pktData(pkt)$

g7 : $pkt \in dom(pktNbHops) \wedge nbh = pktNbHops(pkt)$

g8 : $pkt \notin dom(vPktSeqNo)$

g10 : $pkt \notin dom(vPktFwdr)$
g9 : $pkt \notin dom(vPktSrc)$
g11 : $pkt \notin dom(vPktData)$
g12 : $pkt \notin dom(vPktNbHops)$

then

a1 : $channel := channel \cup \{cn \mapsto pkt\}$
a2 : $vPktSeqNo := vPktSeqNo \cup \{pkt \mapsto sno\}$
a3 : $vPktSrc := vPktSrc \cup \{pkt \mapsto src\}$
a4 : $vPktFwdr := vPktFwdr \cup \{pkt \mapsto fwdr\}$
a5 : $vPktData := vPktData \cup \{pkt \mapsto data\}$
a6 : $vPktNbHops := vPktNbHops \cup \{pkt \mapsto nbh\}$
a7 : $pktSeqNo := \{pkt\} \lhd pktSeqNo$
a8 : $pktSrc := \{pkt\} \lhd pktSrc$
a9 : $pktFwdr := \{pkt\} \lhd pktFwdr$
a10 : $pktData := \{pkt\} \lhd pktData$
a11 : $pktNbHops := \{pkt\} \lhd pktNbHops$

end

Event *find_neighbours* $\widehat{=}$

//(env) channel finds the neighbour list of the forwarder

any

s
nbs
f
pkt

where

g1 : $channel \neq \varnothing$
g2 : $pkt \in ran(channel)$
g3 : $s = initialSrcAddr(pkt)$
g4 : $f \mapsto pkt \in channel$
g5 : $nbs \subseteq ND \wedge nbs = wsnLinks[\{f\}] \wedge nbs \neq \varnothing$

//find all neighbours of current forwarder

g6 : $envNeighbours = \varnothing$

then

a1 : $envNeighbours := envNeighbours \cup (\{pkt\} \times nbs)$

end

Event *assign_forwarder* $\widehat{=}$

//(env) channel assigns specific fowarder - preparing for unicast for later refinement

any

s
nb

pkt
f
nbs

where

g1 : $channel \neq \varnothing$
g2 : $pkt \in ran(channel)$
g3 : $s = initialSrcAddr(pkt)$
g4 : $f \mapsto pkt \in channel$
g5 : $nbs = wsnLinks[\{f\}] \wedge nbs \neq \varnothing$
g6 : $nb \in nbs$ //find all neighbours of current forwarder

then

a1 : $envNeighbours := envNeighbours \cup \{pkt \mapsto nb\}$

end

Event *send_up* $\widehat{=}$

//(env) channel sends the neighbour information

refines *indicate_neighbours*

any

pkt
$nbrs$
f
sno
src
$fwdr$
$data$
nbh

where

g1 : $pkt \in ran(sentDown)$
g2 : $f \mapsto pkt \in sentDown$
g3 : $f \mapsto pkt \notin sentUp$
g4 : $f \mapsto pkt \in channel$
g5 : $nbrs \in \{n | n \in (\mathbb{P}(ND)) \cup \{\{FAILED\_XMIT\}\}\}$
g6 : $nbrs \neq \varnothing$
g7 : $ran(\{pkt\} \lhd envNeighbours) \in (\mathbb{P}(ND) \cup \{\{FAILED\_XMIT\}\})$
g8 : $nbrs = ran(\{pkt\} \lhd envNeighbours) \wedge nbrs \neq \varnothing$

//retrieving neighbours from environment

g9 : $pkt \notin dom(ctlNeighbours)$
g10 : $pkt \in dom(vPktSeqNo) \wedge sno = vPktSeqNo(pkt)$
g11 : $pkt \in dom(vPktSrc) \wedge src = vPktSrc(pkt)$
g12 : $pkt \in dom(vPktFwdr) \wedge fwdr = vPktFwdr(pkt)$
g13 : $pkt \in dom(vPktData) \wedge data = vPktData(pkt)$
g14 : $pkt \in dom(vPktNbHops) \wedge nbh = vPktNbHops(pkt)$

g17 : $pkt \notin dom(pktSeqNo)$
g18 : $pkt \notin dom(pktSrc)$
g19 : $pkt \notin dom(pktFwdr)$
g20 : $pkt \notin dom(pktData)$
g21 : $pkt \notin dom(pktNbHops)$

then

a1 : $ctlNeighbours := ctlNeighbours \cup (\{pkt\} \times nbrs)$
//updating neighbours from environment to controller
a2 : $envNeighbours := \varnothing$
a3 : $channel := channel \setminus \{f \mapsto pkt\}$
a4 : $sentDown := sentDown \setminus \{f \mapsto pkt\}$
a5 : $sentUp := sentUp \cup \{f \mapsto pkt\}$
a6 : $pktSeqNo := pktSeqNo \cup \{pkt \mapsto sno\}$
a7 : $pktSrc := pktSrc \cup \{pkt \mapsto src\}$
a8 : $pktFwdr := pktFwdr \cup \{pkt \mapsto fwdr\}$
a9 : $pktData := pktData \cup \{pkt \mapsto data\}$
a10 : $pktNbHops := pktNbHops \cup \{pkt \mapsto nbh\}$
a11 : $vPktSeqNo := \{pkt\} \lhd vPktSeqNo$
a12 : $vPktSrc := \{pkt\} \lhd vPktSrc$
a13 : $vPktFwdr := \{pkt\} \lhd vPktFwdr$
a14 : $vPktData := \{pkt\} \lhd vPktData$
a15 : $vPktNbHops := \{pkt\} \lhd vPktNbHops$

end

Event *receive_pkt* $\widehat{=}$
//(ctl) each neighbour node receives a packet
refines *receive_pkt*

any

s
nb
pkt
nbh
f

where

g1 : $pkt \in ran(sentUp \cup sentDown)$
g2 : $pkt \mapsto nb \in ctlNeighbours$
g3 : $nb \mapsto pkt \notin ndBuff$
g4 : $nb \notin dom(sentUp \cup sentDown)$
g5 : $s = initialSrcAddr(pkt) \wedge s \neq nb$
g6 : $pkt \in dom(finalDestAddr) \wedge nb \neq finalDestAddr(pkt)$
g7 : $nb \in dom(floodTbl) \wedge pkt \notin floodTbl(nb)$
g8 : $pkt \in dom(pktNbHops) \wedge nbh = pktNbHops(pkt) + 1$

g9 : $pkt \mapsto f \in pktFwdr$

then

a1 : $ndBuff := ndBuff \cup \{nb \mapsto pkt\}$

a2 : $ctlNeighbours := ctlNeighbours \setminus \{pkt \mapsto nb\}$

a3 : $floodTbl(nb) := floodTbl(nb) \cup \{pkt\}$

a4 : $nbHops := nbHops \cup \{nb \mapsto \{pkt \mapsto nbh\}\}$

end

Event *receive_dup_pkt* $\widehat{=}$

//ctl) each neighbour node discards a duplicated packet

extends *receive_dup_pkt*

any

nb

pkt

f

where

g1 : $pkt \in dom(ctlNeighbours)$

g2 : $pkt \mapsto nb \in ctlNeighbours$

g3 : $nb \in dom(floodTbl) \wedge pkt \in floodTbl(nb)$

g4 pkt $\mapsto f \in pktFwdr$

then

a1 : $ctlNeighbours := ctlNeighbours \setminus \{pkt \mapsto nb\}$

end

Event *dest_recv_dataPkt* $\widehat{=}$

//destination receives a data packet

refines *dest_recv_dataPkt*

any

s

des

pkt

$data$

f

where

g1 : $pkt \in PKT$

g2 : $pkt \in dom(finalDestAddr) \wedge des = finalDestAddr(pkt)$

g3 : $des \in dom(destBuff)$

g4 : $pkt \in ran(sentUp \cup sentDown) \setminus destBuff(des)$

g5 : $s = initialSrcAddr(pkt)$

g6 : $\{pkt\} \lhd ctlNeighbours \neq \varnothing$

g7 : $ran(\{pkt\} \lhd ctlNeighbours) \neq \{FAILED\_XMIT\}$

g8 : $des \in (ran(\{pkt\} \lhd ctlNeighbours))$

g9 : $des \in dom(recvedData)$
g10 : $pkt \in dom(pktData) \wedge data = pktData(pkt)$
g11 : $type(pkt) = DATA$
g12 : $pkt \mapsto f \in pktFwdr$

then

a1 : $destBuff(des) := destBuff(des) \cup \{pkt\}$
a2 : $ctlNeighbours := ctlNeighbours \setminus \{pkt \mapsto des\}$
a3 : $floodTbl(des) := floodTbl(des) \cup \{pkt\}$
a4 : $recvedData(des) := recvedData(des) \cup \{data\}$

end

Event *dest_recv_controlPkt* $\widehat{=}$

//(ctl)destination receives a control packet

refines *dest_recv_controlPkt*

any

s
des
pkt
f

where

g1 : $pkt \in PKT$
g2 : $pkt \in dom(finalDestAddr) \wedge des = finalDestAddr(pkt)$
g3 : $des \in dom(destBuff)$
g4 : $pkt \in ran(sentUp \cup sentDown) \setminus destBuff(des)$
g5 : $s = initialSrcAddr(pkt)$
g6 : $\{pkt\} \lhd ctlNeighbours \neq \varnothing$
g7 : $ran(\{pkt\} \lhd ctlNeighbours) \neq \{FAILED\_XMIT\}$
g8 : $des \in (ran(\{pkt\} \lhd ctlNeighbours))$
g9 : $type(pkt) \in CONTROL$
g10 : $pkt \mapsto f \in pktFwdr$

then

a1 : $destBuff(des) := destBuff(des) \cup \{pkt\}$
a2 : $ctlNeighbours := ctlNeighbours \setminus \{pkt \mapsto des\}$
a3 : $floodTbl(des) := floodTbl(des) \cup \{pkt\}$

end

Event *actuating* $\widehat{=}$

//(ctl) alerting emergency: sensed data is above safety threshold

extends *actuating*

any

act
$data$

where

g1 : $act \in Actuators$
g2 : $act \in dom(recvedData)$
g3 : $data \in recvedData(act)$
g4 : $data \geq safetyThreshold$
g5 : $emergencyAlert(act) = FALSE$

then

a1 : $recvedData(act) := recvedData(act) \setminus \{data\}$
a2 : $emergencyAlert(act) := TRUE$

end

Event *no_actuating* $\widehat{=}$

//(ctl) not alerting emergency sensed data is below safety threshold

extends *no_actuating*

any

act
$data$

where

g1 : $act \in Actuators$
g2 : $act \in dom(recvedData)$
g3 : $data \in recvedData(act)$
g4 : $data < safetyThreshold$

then

a1 : $recvedData(act) := recvedData(act) \setminus \{data\}$

end

Event *reset_actuatingStatus* $\widehat{=}$

//(ctl) resetting emergency alert status

extends *reset_actuatingStatus*

any

act

where

g1 : $act \in dom(emergencyAlert)$
g2 : $emergencyAlert(act) = TRUE$

then

a1 : $emergencyAlert(act) := FALSE$

end

Event *finish_tx_pkt* $\widehat{=}$

//(ctl) finishing packet transmission

refines *finish_tx_pkt*

any

pkt
f

where

g1 : $\{pkt\} \lhd ctlNeighbours = \varnothing$
g2 : $f \in ND \wedge f \mapsto pkt \in sentUp$
g3 : $pkt \in ran(sentUp) \wedge pkt \notin ran(sentDown) \wedge pkt \in ran(ndBuff)$

then

a1 : $sentUp := sentUp \setminus \{f \mapsto pkt\}$

end

Event *final_tx_pkt* $\widehat{=}$

//(ctl) the transmitted packet is removed from middleware indicating completion of transmission mechanism

refines *final_tx_pkt*

any

pkt
sk

where

g1 : $pkt \in PKT$
g2 : $pkt \in ran(sentUp) \wedge pkt \notin ran(sentDown) \wedge pkt \notin ran(ndBuff)$
g3 : $pkt \in dom(finalDestAddr) \wedge sk = finalDestAddr(pkt)$
g4 : $sk \in ND$
g5 : $pkt \in destBuff(sk)$
g6 : $\{pkt\} \lhd ctlNeighbours = \varnothing$

then

a1 : $sentUp := sentUp \rhd \{pkt\}$

end

Event *lose_all_neighbours* $\widehat{=}$

//(env) channel loses all neighbour

any

s
nbs
f
pkt

where

g1 : $channel \neq \varnothing$
g2 : $pkt \in ran(channel)$
g3 : $s = initialSrcAddr(pkt)$
g4 : $f \mapsto pkt \in channel$
g5 : $nbs = wsnLinks[\{f\}] \wedge nbs = \varnothing$

//find all neighbours of current forwarder

then

a1 : $envNeighbours := envNeighbours \cup \{pkt \mapsto FAILED\_XMIT\}$

end

Event *lose_specifc_neighbour* $\widehat{=}$

//(env) channel loses a specific neighbour

any

s

nbs

f

pkt

where

g1 : $channel \neq \varnothing$

g2 : $pkt \in ran(channel)$

g3 : $s = initialSrcAddr(pkt)$

g4 : $f \mapsto pkt \in channel$

g5 : $nbs = wsnLinks[\{f\}] \wedge nbs \neq \varnothing$

//find all neighbours of current forwarder

then

a1 : $envNeighbours := envNeighbours \cup \{pkt \mapsto FAILED\_XMIT\}$

end

Event *lose_pkt* $\widehat{=}$

//(ctl) each node records a lost packet

refines *lose_pkt*

any

f

pkt

where

g1 : $pkt \in ran(sentUp) \wedge pkt \notin ran(sentDown)$

g2 : $pkt \notin ran(recLostPkts)$

g3 : $pkt \notin ran(ndBuff)$

g4 : $f \mapsto pkt \in sentUp \wedge f \mapsto pkt \notin sentDown$

g5 : $f \mapsto pkt \notin recLostPkts$

g6 : $ran(\{pkt\} \lhd ctlNeighbours) = \{FAILED\_XMIT\}$

then

a1 : $sentUp := sentUp \rhd \{pkt\}$

a2 : $recLostPkts := recLostPkts \cup \{f \mapsto pkt\}$

a3 : $ctlNeighbours := ctlNeighbours \setminus \{pkt \mapsto FAILED\_XMIT\}$

end

END

C.5 Fourth refinement:Path discover (RREQ flooding)

C.5.1 Context:C4

CONTEXT C4

EXTENDS C3

CONSTANTS

$RREQ$

$RREP$

$RRER$

AXIOMS

axm4_1 : $partition(CONTROL, \{RREQ\}, \{RREP\}, \{RRER\})$ //three control packet types: route request (RREQ), route reply (RREP) and route error (RRER)

END

C.5.2 Machine:M4

MACHINE M4

REFINES M3

SEES C4

VARIABLES

$destBuff$ //(ctl) nodes' receiving buffer

$finalDestAddr$ //(ctl) final destination address

$ndBuff$ //(ctl) nodes' waiting buffer

$sensingNDs$ //(ctl) sensing nodes

$sensedPkts$ //(ctl) sensed packets

$floodedPkts$ //(ctl) flooded packets

$floodTbl$ //(ctl) historical received packets used for detect duplicated packets

$ctlNeighbours$ //(ctl) neighbour lists

$recLostPkts$ //(ctl) record recLostPkts packet at each node

$floodFlg$ //(ctl) flood flag

$envSensedFlg$ //(env) sensing period

$envData$ //(env) environmental/conditional data

$ctlSensedFlg$ //(ctl)sensing period

$senseBuff$ //(ctl) sense buffer

$recvedData$ //(ctl) received data at destination

$emergencyAlert$ //(ctl) actuating status

$pktData$ //(ctl) data attached in transmitted packet

$sentDown$ //(ctl) transmitted packet to channel

$sentUp$ //(ctl) transmitted packet up to channel

$channel$ //(env) channel

$envNeighbours$ //(env) neighbour list

$wsnLinks$ //(env) dynamic WSN links

$crashedLinks$ //(env) crashed WSN Links

$dataSeqNo$ //(ctl) local nodes' current data sequence number at application layer

$floodSeqNo$ (ctl) local nodes' current broadcast sequence number at application layer

$nbHops$ //(ctl) neighbour's hop count

$pktSrc$ //(ctl) src address attached in transmitted packet

$pktSeqNo$ //(ctl) sequence number attached in transmitted packet

$pktFwdr$ //(ctl) fowarder attached in transmitted packet

$pktNbHops$ //(ctl) hop count attached in transmitted packet

$vPktSeqNo$ //(env) virtual packet at channel

$vPktSrc$ //(env) virtual packet at channel

$vPktFwdr$ //(env) virtual packet at channel

$vPktData$ //(env) virtual packet at channel

$vPktNbHops$ //(env) virtual packet at chennel

$bwdRouteTbl$ //(ctl) backwards route table to a source

$bwdNextND$ //(ctl) backwards route table : field next node

$bwdSeqNo$ //(ctl) backwards route table : field sequence number

$bwdHopCnt$ //(ctl) backwards route table : field hop count

$netSeqNo$ //(ctl) link sequence number in a packet

$linkSeqNo$ //(ctl) link sequence number of each node

$updateNbrs$ //(ctl) for update neighbours in the neighbour table

INVARIANTS

inv4_1 : $bwdRouteTbl \in ND \leftrightarrow ND$

inv4_2 : $bwdNextND \in bwdRouteTbl \rightarrow ND$

inv4_3 : $bwdSeqNo \in bwdRouteTbl \rightarrow \mathbb{N}$

inv4_4 : $bwdHopCnt \in bwdRouteTbl \rightarrow \mathbb{N}$

inv4_5 : $netSeqNo \in PKT \rightarrow \mathbb{N}$

inv4_6 : $linkSeqNo \in ND \rightarrow \mathbb{N}$

inv4_7 : $updateNbrs \in ND \leftrightarrow ND$

EVENTS

Initialisation

extended

begin

int0_3 : $destBuff := ND \times \{\varnothing\}$
int0_4 : $finalDestAddr := \varnothing$
int1_1 : $ndBuff := \varnothing$
int1_3 : $sensingNDs := \varnothing$
int1_4 : $sensedPkts := \varnothing$
int1_6 : $floodedPkts := \varnothing$
int1_7 : $floodTbl := ND \times \{\varnothing\}$
int1_8 : $ctlNeighbours := \varnothing$
int1_12 : $floodFlg := ND \times \{FALSE\}$
inv1_20 : $recLostPkts := \varnothing$
int2_1 : $envSensedFlg := (ND \setminus Destination) \times \{FALSE\}$
int2_2 : $envData := (ND \setminus Destination) \times \{0\}$
int2_3 : $ctlSensedFlg := (ND \setminus Destination) \times \{FALSE\}$
int2_4 : $senseBuff := (ND \setminus Destination) \times \{\varnothing\}$
int2_5 : $recvedData := Destination \times \{\varnothing\}$
int2_6 : $emergencyAlert := Actuators \times \{FALSE\}$
int2_7 : $pktData := \varnothing$
int3_1 : $sentUp := \varnothing$
int3_2 : $sentDown := \varnothing$
int3_3 : $channel := \varnothing$
int3_4 : $envNeighbours := \varnothing$
int3_5 : $wsnLinks := \varnothing$
int3_6 : $crashedLinks := \varnothing$
int3_7 : $dataSeqNo := ND \times \{0\}$
int3_8 : $floodSeqNo := ND \times \{0\}$
int3_9 : $nbHops := \varnothing$
int3_10 : $pktSeqNo := \varnothing$
int3_11 : $pktSrc := \varnothing$
int3_12 : $pktFwdr := \varnothing$
int3_13 : $pktNbHops := \varnothing$
int3_14 : $vPktSeqNo := \varnothing$
int3_15 : $vPktSrc := \varnothing$
int3_16 : $vPktFwdr := \varnothing$
int2_8 : $vPktData := \varnothing$
int3_18 : $vPktNbHops := \varnothing$
int4_1 : $bwdRouteTbl := \varnothing$
int4_2 : $bwdSeqNo := \varnothing$
int4_3 : $bwdNextND := \varnothing$
int4_4 : $bwdHopCnt := \varnothing$
int4_5 : $netSeqNo := PKT \times \{0\}$

int4_6 : $linkSeqNo := ND \times \{0\}$
int4_7 : $updateNbrs := \varnothing$
end

Event *add_link* $\widehat{=}$
//(env) channel sets a topology
extends *add_link*
any
x
y
where
g1 : $x \in ND \wedge y \in ND$
g2 : $x \mapsto y \notin wsnLinks$
g3 : $x \neq y$
then
a1 : $wsnLinks := wsnLinks \cup \{x \mapsto y\}$
end

Event *disconnect_link* $\widehat{=}$
//(env) disconnecting a link
extends *disconnect_link*
any
x
y
where
g1 : $x \in ND \wedge y \in ND$
g2 : $x \mapsto y \in wsnLinks$
g3 : $x \mapsto y \notin crashedLinks$
then
a1 : $crashedLinks := crashedLinks \cup \{x \mapsto y\}$
a2 : $wsnLinks := wsnLinks \setminus \{x \mapsto y\}$
end

Event *recover_link* $\widehat{=}$
//(env) recovering a link
extends *recover_link*
any
x
y
where
g1 : $x \in ND \wedge y \in ND$
g2 : $x \mapsto y \in crashedLinks$
g3 : $x \mapsto y \notin wsnLinks$

then

a1 : $wsnLinks := wsnLinks \cup \{x \mapsto y\}$

a2 : $crashedLinks := crashedLinks \setminus \{x \mapsto y\}$

end

Event *start_sensing* $\widehat{=}$

//(env) activating each node to sense data

extends *start_sensing*

any

x

sd

where

g1 : $x \in ND \setminus Destination$

g2 : $envSensedFlg(x) = FALSE$

g3 : $sd \in randomFn$

then

a1 : $envSensedFlg(x) := TRUE$

a2 : $envData(x) := sd$

end

Event *sensing* $\widehat{=}$

//(shd) each node sense a data from environment

extends *sensing*

any

x

sf

sd

where

g1 : $x \in ND \setminus Destination$

g2 : $sf = envSensedFlg(x)$

g3 : $envSensedFlg(x) = TRUE$

g4 : $ctlSensedFlg(x) = FALSE$

g5 : $sd = envData(x)$

then

a1 : $ctlSensedFlg(x) := sf$

a2 : $senseBuff(x) := senseBuff(x) \cup \{sd\}$

end

Event *finish_sensing* $\widehat{=}$

//(shd) finishing sensing data process

extends *finish_sensing*

any

x

where

g1 : $sensingNDs = \{x\}$
g2 : $x \in dom(envSensedFlg)$
g3 : $envSensedFlg(x) = TRUE$
g4 : $ctlSensedFlg(x) = TRUE$
g5 : $ndBuff = \varnothing \wedge (sentDown \cup sentUp) = \varnothing$

then

a1 : $sensingNDs := \varnothing$
a2 : $ctlSensedFlg(x) := FALSE$
a3 : $envSensedFlg(x) := FALSE$

end

Event *create_dataPkt* $\widehat{=}$

//(ctl) creating a data packet

extends *create_dataPkt*

any

s
pkt
$fDes$
$data$
sno
nbh

where

g1 : $pkt \in PKT \wedge s \in ND \setminus Destination$
g2 : $pkt \notin ran(ndBuff \cup sentDown \cup sentUp)$
g3 : $s \notin sensingNDs \wedge pkt \notin (sensedPkts \cup floodedPkts)$
//has not been created yet
g4 : $s = initialSrcAddr(pkt)$
g5 : $s \in dom(floodTbl) \wedge pkt \notin floodTbl(s)$
g6 : $type(pkt) = DATA$
g7 : $fDes \in Destination$
g8 : $pkt \mapsto fDes \notin finalDestAddr$
g9 : $fDes \neq s$
g10 : $ctlSensedFlg(s) = TRUE$
g11 : $s \in dom(senseBuff) \wedge senseBuff \neq \varnothing \wedge data \in senseBuff(s)$
g12 : $pkt \notin dom(pktData)$
g13 : $sno = dataSeqNo(s) + 1 \wedge s \in dom(dataSeqNo)$ //identifing seq number
g14 : $nbh \in \mathbb{Z} \wedge nbh = -1$
g15 : $s \mapsto \{pkt \mapsto nbh\} \notin nbHops$
g16 : $pkt \notin dom(pktSeqNo)$
g17 : $pkt \notin dom(pktSrc)$
g18 : $pkt \notin dom(pktFwdr)$

g19 : $pkt \notin dom(pktNbHops)$

then

a1 : $sensingNDs := sensingNDs \cup \{s\}$

a2 : $sensedPkts := sensedPkts \cup \{pkt\}$

a3 : $ndBuff := ndBuff \cup \{s \mapsto pkt\}$

a4 : $floodTbl(s) := floodTbl(s) \cup \{pkt\}$

a5 : $finalDestAddr := finalDestAddr \cup \{pkt \mapsto fDes\}$

a6 : $senseBuff(s) := senseBuff(s) \setminus \{data\}$

a7 : $pktData := pktData \cup \{pkt \mapsto data\}$

a8 : $dataSeqNo := dataSeqNo \lhd\!\!\!- \{s \mapsto sno\}$

a9 : $nbHops := nbHops \cup \{s \mapsto \{pkt \mapsto nbh\}\}$

a10 : $pktSeqNo := pktSeqNo \lhd\!\!\!- \{pkt \mapsto sno\}$

a11 : $pktSrc := pktSrc \cup \{pkt \mapsto s\}$

a12 : $pktFwdr := pktFwdr \cup \{pkt \mapsto s\}$

a13 : $pktNbHops := pktNbHops \cup \{pkt \mapsto nbh\}$

end

Event *create_controlPkt* $\widehat{=}$

//(ctl) creating a control packet

extends *create_controlPkt*

any

s

pkt

$fDes$

sno

nbh

where

g1 : $pkt \in PKT$

g2 : $pkt \notin ran(ndBuff \cup sentDown \cup sentUp)$

g3 : $pkt \notin (sensedPkts \cup floodedPkts)$

g4 : $s = initialSrcAddr(pkt)$

g5 : $s \in dom(floodTbl) \wedge pkt \notin floodTbl(s)$

g6 : $type(pkt) \in CONTROL$

g7 : $floodFlg(s) = TRUE$

g8 : $fDes \in ND$

g9 : $pkt \mapsto fDes \notin finalDestAddr$

g10 : $fDes \neq s$

g11 : $sno = floodSeqNo(s) + 1 \wedge s \in dom(floodSeqNo)$

g12 : $nbh \in \mathbb{Z} \wedge nbh = -1$

g13 : $s \mapsto \{pkt \mapsto nbh\} \notin nbHops$

g14 : $pkt \notin dom(pktSeqNo)$

g15 : $pkt \notin dom(pktSrc)$

g16 : $pkt \notin dom(pktFwdr)$
g17 : $pkt \notin dom(pktData)$
g18 : $pkt \notin dom(pktNbHops)$

then

a1 : $floodedPkts := floodedPkts \cup \{pkt\}$
a2 : $ndBuff := ndBuff \cup \{s \mapsto pkt\}$
a3 : $floodTbl(s) := floodTbl(s) \cup \{pkt\}$
a4 : $finalDestAddr := finalDestAddr \cup \{pkt \mapsto fDes\}$
a5 : $floodSeqNo := floodSeqNo \lhd\!\!- \{s \mapsto sno\}$
a6 : $nbHops := nbHops \cup \{s \mapsto \{pkt \mapsto nbh\}\}$
a7 : $pktSeqNo := pktSeqNo \lhd\!\!- \{pkt \mapsto sno\}$
a8 : $pktFwdr := pktFwdr \cup \{pkt \mapsto s\}$
a9 : $pktSrc := pktSrc \cup \{pkt \mapsto s\}$
a10 : $pktData := pktData \cup \{pkt \mapsto 0\}$
a11 : $pktNbHops := pktNbHops \cup \{pkt \mapsto nbh\}$

end

Event *start_tx_dataPkt* $\widehat{=}$

//(ctl) starting transmiting a data packet to a channel

extends *start_tx*

any

x
pkt
nbh

where

g1 : $x \mapsto pkt \in ndBuff$
g2 : $x \mapsto pkt \notin sentDown$
g3 : $x \mapsto pkt \notin sentUp$
g4 : $x \mapsto \{pkt \mapsto nbh\} \in nbHops$
g5 : $pkt \in dom(pktFwdr)$
g6 : $pkt \in dom(pktNbHops)$
g7 : $type(pkt) = DATA$

then

a1 : $ndBuff := ndBuff \setminus \{x \mapsto pkt\}$
a2 : $sentDown := sentDown \cup \{x \mapsto pkt\}$
a3 : $pktFwdr := pktFwdr \lhd\!\!- \{pkt \mapsto x\}$
a4 : $pktNbHops := pktNbHops \lhd\!\!- \{pkt \mapsto nbh\}$
a5 : $nbHops := nbHops \setminus \{x \mapsto \{pkt \mapsto nbh\}\}$

end

Event *start_tx_rreq* $\widehat{=}$

(ctl) starting transmiting a route request packet to a channel

extends *start_tx*

any

x

pkt

nbh

$lsno$

where

g1 : $x \mapsto pkt \in ndBuff$

g2 : $x \mapsto pkt \notin sentDown$

g3 : $x \mapsto pkt \notin sentUp$

g4 : $x \mapsto \{pkt \mapsto nbh\} \in nbHops$

g5 : $pkt \in dom(pktFwdr)$

g6 : $pkt \in dom(pktNbHops)$

g7 : $type(pkt) = RREQ$

g8 : $x \in dom(linkSeqNo) \wedge lsno = linkSeqNo(x) + 1$

//calculating the link sequence number from the recent number

then

a1 : $ndBuff := ndBuff \setminus \{x \mapsto pkt\}$

a2 : $sentDown := sentDown \cup \{x \mapsto pkt\}$

a3 : $pktFwdr := pktFwdr \lhd\!\!\!- \{pkt \mapsto x\}$

a4 : $pktNbHops := pktNbHops \lhd\!\!\!- \{pkt \mapsto nbh\}$

a5 : $nbHops := nbHops \setminus \{x \mapsto \{pkt \mapsto nbh\}\}$

a6 : $linkSeqNo := linkSeqNo \lhd\!\!\!- \{x \mapsto lsno\}$

//update the current sequence number of node x

a7 : $netSeqNo(pkt) := lsno$

//update the current sequence number into a flood packet

end

Event *start_tx_controlPkt* $\widehat{=}$

//(ctl) starting transmiting other control packets from buffer to a channel

extends *start_tx*

any

x

pkt

nbh

$lsno$

where

g1 : $x \mapsto pkt \in ndBuff$

g2 : $x \mapsto pkt \notin sentDown$

g3 : $x \mapsto pkt \notin sentUp$

g4 : $x \mapsto \{pkt \mapsto nbh\} \in nbHops$

g5 : $pkt \in dom(pktFwdr)$

g6 : $pkt \in dom(pktNbHops)$

g7 : $type(pkt) \neq RREQ$
g8 : $lsno \in \mathbb{N}$

then

a1 : $ndBuff := ndBuff \setminus \{x \mapsto pkt\}$
a2 : $sentDown := sentDown \cup \{x \mapsto pkt\}$
a3 : $pktFwdr := pktFwdr \lhd\!\!\!- \{pkt \mapsto x\}$
a4 : $pktNbHops := pktNbHops \lhd\!\!\!- \{pkt \mapsto nbh\}$
a5 : $nbHops := nbHops \setminus \{x \mapsto \{pkt \mapsto nbh\}\}$
a6 : $netSeqNo(pkt) := lsno$

end

Event *send_down* $\widehat{=}$

//(shd) sending a packet down to a channel

extends *send_down*

any

cn
pkt
sno
src
$data$
$fwdr$
nbh

where

g1 : $cn \mapsto pkt \in sentDown$
g2 : $cn \mapsto pkt \notin channel$
g3 : $pkt \in dom(pktSeqNo) \wedge sno = pktSeqNo(pkt)$
g4 : $pkt \in dom(pktSrc) \wedge src = pktSrc(pkt)$
g5 : $pkt \in dom(pktFwdr) \wedge fwdr = pktFwdr(pkt)$
g6 : $pkt \in dom(pktData) \wedge data = pktData(pkt)$
g7 : $pkt \in dom(pktNbHops) \wedge nbh = pktNbHops(pkt)$
g8 : $pkt \notin dom(vPktSeqNo)$
g10 : $pkt \notin dom(vPktFwdr)$
g9 : $pkt \notin dom(vPktSrc)$
g11 : $pkt \notin dom(vPktData)$
g12 : $pkt \notin dom(vPktNbHops)$

then

a1 : $channel := channel \cup \{cn \mapsto pkt\}$
a2 : $vPktSeqNo := vPktSeqNo \cup \{pkt \mapsto sno\}$
a3 : $vPktSrc := vPktSrc \cup \{pkt \mapsto src\}$
a4 : $vPktFwdr := vPktFwdr \cup \{pkt \mapsto fwdr\}$
a5 : $vPktData := vPktData \cup \{pkt \mapsto data\}$
a6 : $vPktNbHops := vPktNbHops \cup \{pkt \mapsto nbh\}$

a7 : $pktSeqNo := \{pkt\} \lhd pktSeqNo$
a8 : $pktSrc := \{pkt\} \lhd pktSrc$
a9 : $pktFwdr := \{pkt\} \lhd pktFwdr$
a10 : $pktData := \{pkt\} \lhd pktData$
a11 : $pktNbHops := \{pkt\} \lhd pktNbHops$

end

Event *find_neighbours* $\widehat{=}$

//(env) channel finds the neighbour list of the forwarder

extends *find_neighbour*

any

s
nbs
f
pkt

where

g1 : $channel \neq \varnothing$
g2 : $pkt \in ran(channel)$
g3 : $s = initialSrcAddr(pkt)$
g4 : $f \mapsto pkt \in channel$
g5 : $nbs \subseteq ND \wedge nbs = wsnLinks[\{f\}] \wedge nbs \neq \varnothing$
g6 : $envNeighbours = \varnothing$

then

a1 : $envNeighbours := envNeighbours \cup (\{pkt\} \times nbs)$

end

Event *assign_forwarder* $\widehat{=}$

//(env) channel assigns specific fowarder - preparing for unicast for later refinement

extends *assign_forwarder*

any

s
nb
pkt
f
nbs

where

g1 : $channel \neq \varnothing$
g2 : $pkt \in ran(channel)$
g3 : $s = initialSrcAddr(pkt)$
g4 : $f \mapsto pkt \in channel$
g5 : $nbs = wsnLinks[\{f\}] \wedge nbs \neq \varnothing$

g6 : $nb \in nbs$

then

a1 : $envNeighbours := envNeighbours \cup \{pkt \mapsto nb\}$

end

Event *send_up* $\widehat{=}$

//(env) channel sends the neighbour information

extends *send_up*

any

pkt
$nbrs$
f
sno
src
$fwdr$
$data$
nbh

where

g1 : $pkt \in ran(sentDown)$
g2 : $f \mapsto pkt \in sentDown$
g3 : $f \mapsto pkt \notin sentUp$
g4 : $f \mapsto pkt \in channel$
g5 : $nbrs \in \{n | n \in (\mathbb{P}(ND)) \cup \{\{FAILED\_XMIT\}\}\}$
g6 : $nbrs \neq \varnothing$
g7 : $ran(\{pkt\} \lhd envNeighbours) \in (\mathbb{P}(ND) \cup \{\{FAILED\_XMIT\}\})$
g8 : $nbrs = ran(\{pkt\} \lhd envNeighbours) \wedge nbrs \neq \varnothing$
g9 : $pkt \notin dom(ctlNeighbours)$
g10 : $pkt \in dom(vPktSeqNo) \wedge sno = vPktSeqNo(pkt)$
g11 : $pkt \in dom(vPktSrc) \wedge src = vPktSrc(pkt)$
g12 : $pkt \in dom(vPktFwdr) \wedge fwdr = vPktFwdr(pkt)$
g13 : $pkt \in dom(vPktData) \wedge data = vPktData(pkt)$
g14 : $pkt \in dom(vPktNbHops) \wedge nbh = vPktNbHops(pkt)$
g17 : $pkt \notin dom(pktSeqNo)$
g18 : $pkt \notin dom(pktSrc)$
g19 : $pkt \notin dom(pktFwdr)$
g20 : $pkt \notin dom(pktData)$
g21 : $pkt \notin dom(pktNbHops)$

then

a1 : $ctlNeighbours := ctlNeighbours \cup (\{pkt\} \times nbrs)$
a2 : $envNeighbours := \varnothing$
a3 : $channel := channel \setminus \{f \mapsto pkt\}$
a4 : $sentDown := sentDown \setminus \{f \mapsto pkt\}$

a5 : $sentUp := sentUp \cup \{f \mapsto pkt\}$
a6 : $pktSeqNo := pktSeqNo \cup \{pkt \mapsto sno\}$
a7 : $pktSrc := pktSrc \cup \{pkt \mapsto src\}$
a8 : $pktFwdr := pktFwdr \cup \{pkt \mapsto fwdr\}$
a9 : $pktData := pktData \cup \{pkt \mapsto data\}$
a10 : $pktNbHops := pktNbHops \cup \{pkt \mapsto nbh\}$
a11 : $vPktSeqNo := \{pkt\} \lhd vPktSeqNo$
a12 : $vPktSrc := \{pkt\} \lhd vPktSrc$
a13 : $vPktFwdr := \{pkt\} \lhd vPktFwdr$
a14 : $vPktData := \{pkt\} \lhd vPktData$
a15 : $vPktNbHops := \{pkt\} \lhd vPktNbHops$

end

Event *receive_dataPkt* $\widehat{=}$

//(ctl) each neighbour node receives a data packet from channel

extends *receive_pkt*

any

s
nb
pkt
nbh
f

where

g1 : $pkt \in ran(sentUp \cup sentDown)$
g2 : $pkt \mapsto nb \in ctlNeighbours$
g3 : $nb \mapsto pkt \notin ndBuff$
g4 : $nb \notin dom(sentUp \cup sentDown)$
g5 : $s = initialSrcAddr(pkt) \wedge s \neq nb$
g6 : $pkt \in dom(finalDestAddr) \wedge nb \neq finalDestAddr(pkt)$
g7 : $nb \in dom(floodTbl) \wedge pkt \notin floodTbl(nb)$
g8 : $pkt \in dom(pktNbHops) \wedge nbh = pktNbHops(pkt) + 1$
g9 : $pkt \mapsto f \in pktFwdr$
g10 : $type(pkt) = DATA$

then

a1 : $ndBuff := ndBuff \cup \{nb \mapsto pkt\}$
a2 : $ctlNeighbours := ctlNeighbours \setminus \{pkt \mapsto nb\}$
a3 : $floodTbl(nb) := floodTbl(nb) \cup \{pkt\}$
a4 : $nbHops := nbHops \cup \{nb \mapsto \{pkt \mapsto nbh\}\}$

end

Event *receive_dup_dataPkt* $\widehat{=}$

//(ctl) each neighbour node including sink receives a duplicated data packet from channel

extends *receive_dup_pkt*

any

nb

pkt

f

where

g1 : $pkt \in dom(ctlNeighbours)$

g2 : $pkt \mapsto nb \in ctlNeighbours$

g3 : $nb \in dom(floodTbl) \wedge pkt \in floodTbl(nb)$

g4 : $pkt \mapsto f \in pktFwdr$

g5 : $type(pkt) = DATA$

then

a1 : $ctlNeighbours := ctlNeighbours \setminus \{pkt \mapsto nb\}$

end

Event *receive_controlPkt* $\widehat{=}$

//(ctl) each neighbour node receives a control packet from a channel

extends *receive_pkt*

any

s

nb

pkt

nbh

f

where

g1 : $pkt \in ran(sentUp \cup sentDown)$

g2 : $pkt \mapsto nb \in ctlNeighbours$

g3 : $nb \mapsto pkt \notin ndBuff$

g4 : $nb \notin dom(sentUp \cup sentDown)$

g5 : $s = initialSrcAddr(pkt) \wedge s \neq nb$

g6 : $pkt \in dom(finalDestAddr) \wedge nb \neq finalDestAddr(pkt)$

g7 : $nb \in dom(floodTbl) \wedge pkt \notin floodTbl(nb)$

g8 : $pkt \in dom(pktNbHops) \wedge nbh = pktNbHops(pkt) + 1$

g9 : $pkt \mapsto f \in pktFwdr$

g10 : $type(pkt) \in CONTROL$

g11 : $f \in ND \wedge nb \in ND$

g12 : $f \mapsto nb \notin updateNbrs$

then

a1 : $ndBuff := ndBuff \cup \{nb \mapsto pkt\}$

a2 : $ctlNeighbours := ctlNeighbours \setminus \{pkt \mapsto nb\}$

a3 : $floodTbl(nb) := floodTbl(nb) \cup \{pkt\}$

a4 : $nbHops := nbHops \cup \{nb \mapsto \{pkt \mapsto nbh\}\}$
a5 : $updateNbrs := updateNbrs \cup \{f \mapsto nb\}$

end

Event *receive_dup_controlPkt* $\widehat{=}$

//(ctl) each neighbour node including sink receives a duplicated control packet from channel

extends *receive_dup_pkt*

any

nb
pkt
f

where

g1 : $pkt \in dom(ctlNeighbours)$
g2 : $pkt \mapsto nb \in ctlNeighbours$

g3 : $nb \in dom(floodTbl) \wedge pkt \in floodTbl(nb)$
g4 : $pkt \mapsto f \in pktFwdr$
g5 : $type(pkt) \neq DATA$
g6 : $f \in ND \wedge nb \in ND$
g7 : $f \mapsto nb \notin updateNbrs$

then

a1 : $ctlNeighbours := ctlNeighbours \setminus \{pkt \mapsto nb\}$

end

Event *dest_recv_dataPkt* $\widehat{=}$

//(ctl) destination receives a data packet from a channel

extends *dest_recv_dataPkt*

any

s
des
pkt
$data$
f

where

g1 : $pkt \in PKT$
g2 : $pkt \in dom(finalDestAddr) \wedge des = finalDestAddr(pkt)$
g3 : $des \in dom(destBuff)$
g4 : $pkt \in ran(sentUp \cup sentDown) \setminus destBuff(des)$
g5 : $s = initialSrcAddr(pkt)$
g6 : $\{pkt\} \lhd ctlNeighbours \neq \varnothing$
g7 : $ran(\{pkt\} \lhd ctlNeighbours) \neq \{FAILED\_XMIT\}$

g8 : $des \in (ran(\{pkt\} \lhd ctlNeighbours))$
g9 : $des \in dom(recvedData)$
g10 : $pkt \in dom(pktData) \land data = pktData(pkt)$
g11 : $type(pkt) = DATA$
g12 : $pkt \mapsto f \in pktFwdr$

then

a1 : $destBuff(des) := destBuff(des) \cup \{pkt\}$
a2 : $ctlNeighbours := ctlNeighbours \setminus \{pkt \mapsto des\}$
a3 : $floodTbl(des) := floodTbl(des) \cup \{pkt\}$
a4 : $recvedData(des) := recvedData(des) \cup \{data\}$

end

Event *dest_recv_controlPkt* $\widehat{=}$

//(ctl) destination receives a control packet from a channel

extends *dest_recv_controlPkt*

any

s
des
pkt
f

where

g1 : $pkt \in PKT$
g2 : $pkt \in dom(finalDestAddr) \land des = finalDestAddr(pkt)$
g3 : $des \in dom(destBuff)$
g4 : $pkt \in ran(sentUp \cup sentDown) \setminus destBuff(des)$
g5 : $s = initialSrcAddr(pkt)$
g6 : $\{pkt\} \lhd ctlNeighbours \neq \varnothing$
g7 : $ran(\{pkt\} \lhd ctlNeighbours) \neq \{FAILED\_XMIT\}$
g8 : $des \in (ran(\{pkt\} \lhd ctlNeighbours))$
g9 : $type(pkt) \in CONTROL$
g10 : $pkt \mapsto f \in pktFwdr$
g11 : $type(pkt) \neq DATA$
g12 : $f \in ND \land des \in ND$
g13 : $f \mapsto des \notin updateNbrs$

then

a1 : $destBuff(des) := destBuff(des) \cup \{pkt\}$
a2 : $ctlNeighbours := ctlNeighbours \setminus \{pkt \mapsto des\}$
a3 : $floodTbl(des) := floodTbl(des) \cup \{pkt\}$
a4 : $updateNbrs := updateNbrs \cup \{f \mapsto des\}$

end

Event *actuating* $\widehat{=}$

//(ctl) alerting emergency: sensed data is above safety threshold

extends *actuating*

any

act

$data$

where

g1 : $act \in Actuators$

g2 : $act \in dom(recvedData)$

g3 : $data \in recvedData(act)$

g4 : $data \geq safetyThreshold$

g5 : $emergencyAlert(act) = FALSE$

then

a1 : $recvedData(act) := recvedData(act) \setminus \{data\}$

a2 : $emergencyAlert(act) := TRUE$

end

Event *no_actuating* $\widehat{=}$

//(ctl) not alerting emergency sensed data is below safety threshold

extends *no_actuating*

any

act

$data$

where

g1 : $act \in Actuators$

g2 : $act \in dom(recvedData)$

g3 : $data \in recvedData(act)$

g4 : $data < safetyThreshold$

then

a1 : $recvedData(act) := recvedData(act) \setminus \{data\}$

end

Event *reset_actuatingStatus* $\widehat{=}$

//(ctl) resetting emergency alert status

extends *reset_actuatingStatus*

any

act

where

g1 : $act \in dom(emergencyAlert)$

g2 : $emergencyAlert(act) = TRUE$

then

a1 : $emergencyAlert(act) := FALSE$

end

Event *add_routeEntry* $\widehat{=}$

//(ctl) add new entry into the route table

any

x

y

s

pkt

sNo

$hCnt$

where

g1 : $updateNbrs \neq \varnothing$

g2 : $x \in ND \wedge y \in ND \wedge x \mapsto y \in updateNbrs$

g3 : $pkt \in dom(initialSrcAddr) \wedge s = initialSrcAddr(pkt)$

g4 : $y \mapsto s \notin bwdRouteTbl$

g5 : $pkt \mapsto x \in pktFwdr$

g6 : $sNo = netSeqNo(pkt) \wedge y \mapsto x \mapsto sNo \notin bwdSeqNo$

g7 : $hCnt \in \mathbb{N} \wedge pkt \in dom(pktNbHops) \wedge hCnt = pktNbHops(pkt) + 1$
$\wedge\ y \mapsto x \mapsto hCnt \notin bwdHopCnt$

g8 : $ctlNeighbours = \varnothing$

g9 : $type(pkt) = RREQ$

then

a1 : $bwdRouteTbl := bwdRouteTbl \cup \{y \mapsto s\}$

a2 : $bwdNextND := bwdNextND \cup \{y \mapsto s \mapsto x\}$

a3 : $bwdSeqNo := bwdSeqNo \cup \{y \mapsto s \mapsto sNo\}$

a4 : $bwdHopCnt := bwdHopCnt \cup \{y \mapsto s \mapsto hCnt\}$

a5 : $updateNbrs := updateNbrs \setminus \{x \mapsto y\}$

end

Event *add_routeEntry2* $\widehat{=}$

//(ctl) add new entry into the route table

any

x

y

s

pkt

sNo

$hCnt$

where

g1 : $updateNbrs \neq \varnothing$

//there is some nodes that have to update thier neighbour table

g2 : $x \in ND \wedge y \in ND \wedge x \mapsto y \in updateNbrs$

g3 : $pkt \in dom(initialSrcAddr) \wedge s = initialSrcAddr(pkt)$

g5 : $pkt \mapsto x \in pktFwdr$

g6 : $sNo = netSeqNo(pkt) \wedge y \mapsto x \mapsto sNo \notin bwdSeqNo$

g7 : $hCnt \in \mathbb{N} \wedge pkt \in dom(pktNbHops) \wedge hCnt = pktNbHops(pkt) + 1$
$\wedge\, y \mapsto x \mapsto hCnt \notin bwdHopCnt$
g8 : $ctlNeighbours = \varnothing$
g9 : $type(pkt) \neq DATA \wedge type(pkt) \neq RREQ$

then

a1 : $updateNbrs := updateNbrs \setminus \{x \mapsto y\}$

end

Event *add_routeEntry3* $\widehat{=}$

//(ctl) add new entry into the route table

any

x
y
s
pkt
sNo
$hCnt$

where

g1 : $updateNbrs \neq \varnothing$

//there is some nodes that have to update thier neighbour table

g2 : $x \in ND \wedge y \in ND \wedge x \mapsto y \in updateNbrs$
g3 : $pkt \in dom(initialSrcAddr) \wedge s = initialSrcAddr(pkt)$
g5 : $pkt \mapsto x \in pktFwdr$
g6 : $sNo = netSeqNo(pkt) \wedge y \mapsto x \mapsto sNo \notin bwdSeqNo$
g7 : $hCnt \in \mathbb{N} \wedge pkt \in dom(pktNbHops) \wedge hCnt = pktNbHops(pkt) + 1$
$\wedge\, y \mapsto x \mapsto hCnt \notin bwdHopCnt$
g8 : $ctlNeighbours = \varnothing$
g9 : $type(pkt) \neq DATA \wedge type(pkt) \neq RREQ$

then

a1 : $updateNbrs := updateNbrs \setminus \{x \mapsto y\}$

end

Event *finish_tx_pkt* $\widehat{=}$

//(ctl)finishing packet transmission

extends *finish_tx_pkt*

any

pkt
f

where

g1 : $\{pkt\} \lhd ctlNeighbours = \varnothing$
g2 : $f \in ND \wedge f \mapsto pkt \in sentUp$
g3 : $pkt \in ran(sentUp) \wedge pkt \notin ran(sentDown) \wedge pkt \in ran(ndBuff)$

then

a1 : $sentUp := sentUp \setminus \{f \mapsto pkt\}$

end

Event *final_tx_pkt* $\widehat{=}$

//(ctl) the transmitted packet is removed from middleware indicating completion of transmission mechanism

extends *final_tx_pkt*

any

pkt

sk

where

g1 : $pkt \in PKT$

g2 : $pkt \in ran(sentUp) \wedge pkt \notin ran(sentDown) \wedge pkt \notin ran(ndBuff)$

g3 : $pkt \in dom(finalDestAddr) \wedge sk = finalDestAddr(pkt)$

g4 : $sk \in ND$

g5 : $pkt \in destBuff(sk)$

g6 : $\{pkt\} \lhd ctlNeighbours = \varnothing$

then

a1 : $sentUp := sentUp \rhd \{pkt\}$

end

Event *lose_all_neighbours* $\widehat{=}$

//(env) channel loses all neighbour

extends *lose_all_neighbours*

any

s

nbs

f

pkt

where

g1 : $channel \neq \varnothing$

g2 : $pkt \in ran(channel)$

g3 : $s = initialSrcAddr(pkt)$

g4 : $f \mapsto pkt \in channel$

g5 : $nbs = wsnLinks[\{f\}] \wedge nbs = \varnothing$

then

a1 : $envNeighbours := envNeighbours \cup \{pkt \mapsto FAILED\_XMIT\}$

end

Event *lose_specifc_neighbour* $\widehat{=}$

//(env) channel loses a specific neighbour

extends *lose_specifc_neighbour*

any

s

nbs
f
pkt
where

g1 : $channel \neq \varnothing$
g2 : $pkt \in ran(channel)$
g3 : $s = initialSrcAddr(pkt)$
g4 : $f \mapsto pkt \in channel$
g5 : $nbs = wsnLinks[\{f\}] \wedge nbs \neq \varnothing$
then

a1 : $envNeighbours := envNeighbours \cup \{pkt \mapsto FAILED\_XMIT\}$
end

Event *lose_pkt* $\widehat{=}$
//(ctl) each node records a lost packet
extends *lose_pkt*
any

f
pkt
where

g1 : $pkt \in ran(sentUp) \wedge pkt \notin ran(sentDown)$
g2 : $pkt \notin ran(recLostPkts)$
g3 : $pkt \notin ran(ndBuff)$
g4 : $f \mapsto pkt \in sentUp \wedge f \mapsto pkt \notin sentDown$
g5 : $f \mapsto pkt \notin recLostPkts$
g6 : $ran(\{pkt\} \lhd ctlNeighbours) = \{FAILED\_XMIT\}$
then

a1 : $sentUp := sentUp \rhd \{pkt\}$
a2 : $recLostPkts := recLostPkts \cup \{f \mapsto pkt\}$
a3 : $ctlNeighbours := ctlNeighbours \setminus \{pkt \mapsto FAILED\_XMIT\}$
end

END

C.6 Fifth refinement:Forward path setup (RREP unicasting) and packet unicast forwarding mechanism

C.6.1 Machine:M5

MACHINE M5
REFINES M4
SEES C4
VARIABLES

$destBuff$ //(ctl) nodes' receiving buffer
$finalDestAddr$ //(ctl) final destination address
$ndBuff$ //(ctl) nodes' waiting buffer
$sensingNDs$ //(ctl) sensing nodes
$sensedPkts$ //(ctl) sensed packets
$floodedPkts$ //(ctl) flooded packets
$floodTbl$ //(ctl) historical received packets used for detect duplicated packets
$ctlNeighbours$ //(ctl) neighbour lists
$recLostPkts$ //(ctl) record recLostPkts packet at each node
$floodFlg$ //(ctl) flood flag
$envSensedFlg$ //(env) sensing period
$envData$ //(env) environmental/conditional data
$ctlSensedFlg$ //(ctl) sensing period
$senseBuff$ //(ctl) sense buffer
$recvedData$ //(ctl) received data at destination
$emergencyAlert$ //(ctl) actuating status
$pktData$ //(ctl) data attached in transmitted packet
$sentDown$ //(ctl) transmitted packet to channel
$sentUp$ //(ctl) transmitted packet up from channel
$channel$ //(env) channel
$envNeighbours$ //(env) neighbour list
$wsnLinks$ //(env) dynamic WSN links
$crashedLinks$ (env) crashed WSN Links
$dataSeqNo$
//(ctl) local nodes' current data sequence number at application layer
$floodSeqNo$
//(ctl) local nodes' current broadcast sequence number at application layer
$nbHops$ //(ctl) neighbour's hop count
$pktSrc$ //(ctl) src address attached in transmitted packet
$pktSeqNo$ //(ctl) sequence number attached in transmitted packet
$pktFwdr$ //(ctl) fowarder attached in transmitted packet
$pktNbHops$ //(ctl) hop count attached in transmitted packet
$vPktSeqNo$ //(env) virtual packet at channel
$vPktSrc$ //(env) virtual packet at channel
$vPktFwdr$ //(env) virtual packet at channel
$vPktData$ //(env) virtual packet at channel
$vPktNbHops$ //(env) virtual packet at channel
$updateNbrs$ //(ctl) for update neighbours in the neighbour table
$bwdRouteTbl$ //(ctl) backwards route table to a source
$bwdNextND$ //(ctl) neighbour table: field next node
$bwdSeqNo$ //(ctl) neighbour table: field sequence number
$bwdHopCnt$ //(ctl) neighbour table: field hop count

$fwdRouteTbl$ //(ctl) forwards route table to a source
$fwdNextND$ //(ctl) forwards route table: field next node
$fwdSeqNo$ //(ctl) forwards route table: field:sequence no in neighbour table
$fwdHopCnt$ //(ctl) forwards route table: field hop count in neighbour table
$netSeqNo$ //(ctl) link sequence number in a packet
$linkSeqNo$ //(ctl) link sequence number of each node
$rrepLists$ //(ctl) route reply lists
$rrepSeqNo$ //(ctl) sequence of route reply at each node
$netDestAddr$ //(ctl) destination address of rrep
$envDestAddr$ //(ctl) destination addr for a channel
$rrepFlg$ //(ctl) rrep flag

INVARIANTS

inv4_5 : $fwdRouteTbl \in ND \leftrightarrow ND$
inv4_6 : $fwdNextND \in fwdRouteTbl \rightarrow ND$
inv4_7 : $fwdSeqNo \in fwdRouteTbl \rightarrow \mathbb{N}$
inv4_8 : $fwdHopCnt \in fwdRouteTbl \rightarrow \mathbb{N}$
inv4_9 : $rrepLists \in ND \leftrightarrow PKT$
inv4_10 : $rrepSeqNo \in ND \rightarrow \mathbb{N}$
inv4_11 : $netDestAddr \in PKT \nrightarrow (ND \cup \{BROADCAST\})$
inv4_12 : $envDestAddr \in PKT \nrightarrow (ND \cup \{BROADCAST\})$
inv4_14 : $rrepFlg \in ND \rightarrow BOOL$

EVENTS

Initialisation

extended

begin

int0_3 : $destBuff := ND \times \{\varnothing\}$
int0_4 : $finalDestAddr := \varnothing$
int1_1 : $ndBuff := \varnothing$
int1_3 : $sensingNDs := \varnothing$
int1_4 : $sensedPkts := \varnothing$
int1_6 : $floodedPkts := \varnothing$
int1_7 : $floodTbl := ND \times \{\varnothing\}$
int1_8 : $ctlNeighbours := \varnothing$
int1_12 : $floodFlg := ND \times \{FALSE\}$
inv1_20 : $recLostPkts := \varnothing$
int2_1 : $envSensedFlg := (ND \setminus Destination) \times \{FALSE\}$
int2_2 : $envData := (ND \setminus Destination) \times \{0\}$
int2_3 : $ctlSensedFlg := (ND \setminus Destination) \times \{FALSE\}$
int2_4 : $senseBuff := (ND \setminus Destination) \times \{\varnothing\}$
int2_5 : $recvedData := Destination \times \{\varnothing\}$
int2_6 : $emergencyAlert := Actuators \times \{FALSE\}$

int2_7 : $pktData := \varnothing$
int3_1 : $sentUp := \varnothing$
int3_2 : $sentDown := \varnothing$
int3_3 : $channel := \varnothing$
int3_4 : $envNeighbours := \varnothing$
int3_5 : $wsnLinks := \varnothing$
int3_6 : $crashedLinks := \varnothing$
int3_7 : $dataSeqNo := ND \times \{0\}$
int3_8 : $floodSeqNo := ND \times \{0\}$
int3_9 : $nbHops := \varnothing$
int3_10 : $pktSeqNo := \varnothing$
int3_11 : $pktSrc := \varnothing$
int3_12 : $pktFwdr := \varnothing$
int3_13 : $pktNbHops := \varnothing$
int3_14 : $vPktSeqNo := \varnothing$
int3_15 : $vPktSrc := \varnothing$
int3_16 : $vPktFwdr := \varnothing$
int2_8 : $vPktData := \varnothing$
int3_18 : $vPktNbHops := \varnothing$
int4_1 : $bwdRouteTbl := \varnothing$
int4_2 : $bwdSeqNo := \varnothing$
int4_3 : $bwdNextND := \varnothing$
int4_4 : $bwdHopCnt := \varnothing$
int4_5 : $netSeqNo := PKT \times \{0\}$
int4_6 : $linkSeqNo := ND \times \{0\}$
int4_7 : $updateNbrs := \varnothing$
int5_1 : $fwdRouteTbl := \varnothing$
int5_2 : $fwdSeqNo := \varnothing$
int5_3 : $fwdNextND := \varnothing$
int5_4 : $fwdHopCnt := \varnothing$
int5_5 : $rrepLists := \varnothing$
int5_6 : $rrepSeqNo := ND \times \{0\}$
int5_7 : $netDestAddr := \varnothing$
int5_8 : $envDestAddr := \varnothing$
int5_9 : $rrepFlg := ND \times \{FALSE\}$

end

Event *add_link* $\widehat{=}$

//(env) channel sets a topology

extends *add_link*

any

x

y

where

g1 : $x \in ND \wedge y \in ND$

g2 : $x \mapsto y \notin wsnLinks$

g3 : $x \neq y$

then

a1 : $wsnLinks := wsnLinks \cup \{x \mapsto y\}$

end

Event *disconnect_link* $\widehat{=}$

//(env) disconnecting a link

extends *disconnect_link*

any

x

y

where

g1 : $x \in ND \wedge y \in ND$

g2 : $x \mapsto y \in wsnLinks$

g3 : $x \mapsto y \notin crashedLinks$

then

a1 : $crashedLinks := crashedLinks \cup \{x \mapsto y\}$

a2 : $wsnLinks := wsnLinks \setminus \{x \mapsto y\}$

end

Event *recover_link* $\widehat{=}$

//(env) recovering a link

extends *recover_link*

any

x

y

where

g1 : $x \in ND \wedge y \in ND$

g2 : $x \mapsto y \in crashedLinks$

g3 : $x \mapsto y \notin wsnLinks$

then

a1 : $wsnLinks := wsnLinks \cup \{x \mapsto y\}$

a2 : $crashedLinks := crashedLinks \setminus \{x \mapsto y\}$

end

Event *start_sensing* $\widehat{=}$

//(env) activating each node to sense data

extends *start_sensing*

any

x

sd

where

g1 : $x \in ND \setminus Destination$

g2 : $envSensedFlg(x) = FALSE$

g3 : $sd \in randomFn$

then

a1 : $envSensedFlg(x) := TRUE$

a2 : $envData(x) := sd$

end

Event *sensing* $\widehat{=}$

//(shd) each node sense a data from environment

extends *sensing*

any

x

sf

sd

where

g1 : $x \in ND \setminus Destination$

g2 : $sf = envSensedFlg(x)$

g3 : $envSensedFlg(x) = TRUE$

g4 : $ctlSensedFlg(x) = FALSE$

g5 : $sd = envData(x)$

then

a1 : $ctlSensedFlg(x) := sf$

a2 : $senseBuff(x) := senseBuff(x) \cup \{sd\}$

end

Event *finish_sensing* $\widehat{=}$

//(shd) finishing sensing data process

extends *finish_sensing*

any

x

where

g1 : $sensingNDs = \{x\}$

g2 : $x \in dom(envSensedFlg)$

g3 : $envSensedFlg(x) = TRUE$

g4 : $ctlSensedFlg(x) = TRUE$

g5 : $ndBuff = \varnothing \wedge (sentDown \cup sentUp) = \varnothing$

then

a1 : $sensingNDs := \varnothing$

a2 : $ctlSensedFlg(x) := FALSE$

a3 : $envSensedFlg(x) := FALSE$

@a3 sensingNDs := $\varnothing$

end

Event *create_dataPkt* $\widehat{=}$

//creating a data packet

extends *create_dataPkt*

any

s

pkt

$fDes$

$data$

sno

nbh

where

g1 : $pkt \in PKT \wedge s \in ND \setminus Destination$

g2 : $pkt \notin ran(ndBuff \cup sentDown \cup sentUp)$

g3 : $s \notin sensingNDs \wedge pkt \notin (sensedPkts \cup floodedPkts)$

g4 : $s = initialSrcAddr(pkt)$

g5 : $s \in dom(floodTbl) \wedge pkt \notin floodTbl(s)$

g6 : $type(pkt) = DATA$

g7 : $fDes \in Destination$

g8 : $pkt \mapsto fDes \notin finalDestAddr$

g9 : $fDes \neq s$

g10 : $ctlSensedFlg(s) = TRUE$

g11 : $s \in dom(senseBuff) \wedge senseBuff \neq \varnothing \wedge data \in senseBuff(s)$

g12 : $pkt \notin dom(pktData)$

g13 : $sno = dataSeqNo(s) + 1 \wedge s \in dom(dataSeqNo)$

g14 : $nbh \in \mathbb{Z} \wedge nbh = -1$

g15 : $s \mapsto \{pkt \mapsto nbh\} \notin nbHops$

g16 : $pkt \notin dom(pktSeqNo)$

g17 : $pkt \notin dom(pktSrc)$

g18 : $pkt \notin dom(pktFwdr)$

g19 : $pkt \notin dom(pktNbHops)$

g20 : $pkt \notin dom(netDestAddr)$

then

a1 : $sensingNDs := sensingNDs \cup \{s\}$

a2 : $sensedPkts := sensedPkts \cup \{pkt\}$

a3 : $ndBuff := ndBuff \cup \{s \mapsto pkt\}$

a4 : $floodTbl(s) := floodTbl(s) \cup \{pkt\}$

a5 : $finalDestAddr := finalDestAddr \cup \{pkt \mapsto fDes\}$

a6 : $senseBuff(s) := senseBuff(s) \setminus \{data\}$

a7 : $pktData := pktData \cup \{pkt \mapsto data\}$
a8 : $dataSeqNo := dataSeqNo \lhd\!\!\!- \{s \mapsto sno\}$
a9 : $nbHops := nbHops \cup \{s \mapsto \{pkt \mapsto nbh\}\}$
a10 : $pktSeqNo := pktSeqNo \lhd\!\!\!- \{pkt \mapsto sno\}$
a11 : $pktSrc := pktSrc \cup \{pkt \mapsto s\}$
a12 : $pktFwdr := pktFwdr \cup \{pkt \mapsto s\}$
a13 : $pktNbHops := pktNbHops \cup \{pkt \mapsto nbh\}$
a14 : $netDestAddr := netDestAddr \cup \{pkt \mapsto BROADCAST\}$

end

Event *create_rreqPkt* $\widehat{=}$

//(ctl) creating a RREQ packet

extends *create_controlPkt*

any

s
pkt
$fDes$
sno
nbh

where

g1 : $pkt \in PKT$
g2 : $pkt \notin ran(ndBuff \cup sentDown \cup sentUp)$
g3 : $pkt \notin (sensedPkts \cup floodedPkts)$
g4 : $s = initialSrcAddr(pkt)$
g5 : $s \in dom(floodTbl) \wedge pkt \notin floodTbl(s)$
g6 : $type(pkt) \in CONTROL$
g7 : $floodFlg(s) = TRUE$
g8 : $fDes \in ND$
g9 : $pkt \mapsto fDes \notin finalDestAddr$
g10 : $fDes \neq s$
g11 : $sno = floodSeqNo(s) + 1 \wedge s \in dom(floodSeqNo)$
g12 : $nbh \in \mathbb{Z} \wedge nbh = -1$
g13 : $s \mapsto \{pkt \mapsto nbh\} \notin nbHops$
g14 : $pkt \notin dom(pktSeqNo)$
g15 : $pkt \notin dom(pktSrc)$
g16 : $pkt \notin dom(pktFwdr)$
g17 : $pkt \notin dom(pktData)$
g18 : $pkt \notin dom(pktNbHops)$
g19 : $type(pkt) = RREQ$
g20 : $pkt \notin dom(netDestAddr)$
g21 : $rrepFlg(s) = FALSE$

then

a1 : $floodedPkts := floodedPkts \cup \{pkt\}$
a2 : $ndBuff := ndBuff \cup \{s \mapsto pkt\}$
a3 : $floodTbl(s) := floodTbl(s) \cup \{pkt\}$
a4 : $finalDestAddr := finalDestAddr \cup \{pkt \mapsto fDes\}$
a5 : $floodSeqNo := floodSeqNo \Leftarrow \{s \mapsto sno\}$
a6 : $nbHops := nbHops \cup \{s \mapsto \{pkt \mapsto nbh\}\}$
a7 : $pktSeqNo := pktSeqNo \Leftarrow \{pkt \mapsto sno\}$
a8 : $pktFwdr := pktFwdr \cup \{pkt \mapsto s\}$
a9 : $pktSrc := pktSrc \cup \{pkt \mapsto s\}$
a10 : $pktData := pktData \cup \{pkt \mapsto 0\}$
a11 : $pktNbHops := pktNbHops \cup \{pkt \mapsto nbh\}$
a12 : $netDestAddr := netDestAddr \cup \{pkt \mapsto BROADCAST\}$

end

Event *create_rrepPkt* $\widehat{=}$

//(ctl) create a RREP packet

extends *create_controlPkt*

any

s
pkt
$fDes$
sno
nbh
$rreq$

where

g1 : $pkt \in PKT$
g2 : $pkt \notin ran(ndBuff \cup sentDown \cup sentUp)$
g3 : $pkt \notin (sensedPkts \cup floodedPkts)$
g4 : $s = initialSrcAddr(pkt)$
g5 : $s \in dom(floodTbl) \wedge pkt \notin floodTbl(s)$
g6 : $type(pkt) \in CONTROL$
g7 : $floodFlg(s) = TRUE$
g8 : $fDes \in ND$
g9 : $pkt \mapsto fDes \notin finalDestAddr$
g10 : $fDes \neq s$
g11 : $sno = floodSeqNo(s) + 1 \wedge s \in dom(floodSeqNo)$
g12 : $nbh \in \mathbb{Z} \wedge nbh = -1$
g13 : $s \mapsto \{pkt \mapsto nbh\} \notin nbHops$
g14 : $pkt \notin dom(pktSeqNo)$
g15 : $pkt \notin dom(pktSrc)$
g16 : $pkt \notin dom(pktFwdr)$
g17 : $pkt \notin dom(pktData)$

g18 : $pkt \notin dom(pktNbHops)$
g19 : $type(pkt) = RREP$
g20 : $s \mapsto rreq \in rrepLists$
g21 : $pkt \notin dom(netDestAddr)$
g22 : $fDes = initialSrcAddr(rreq)$
g23 : $rrepFlg(s) = TRUE$

then

a1 : $floodedPkts := floodedPkts \cup \{pkt\}$
a2 : $ndBuff := ndBuff \cup \{s \mapsto pkt\}$
a3 : $floodTbl(s) := floodTbl(s) \cup \{pkt\}$
a4 : $finalDestAddr := finalDestAddr \cup \{pkt \mapsto fDes\}$
a5 : $floodSeqNo := floodSeqNo \lhd\!\!\!- \{s \mapsto sno\}$
a6 : $nbHops := nbHops \cup \{s \mapsto \{pkt \mapsto nbh\}\}$
a7 : $pktSeqNo := pktSeqNo \lhd\!\!\!- \{pkt \mapsto sno\}$
a8 : $pktFwdr := pktFwdr \cup \{pkt \mapsto s\}$
a9 : $pktSrc := pktSrc \cup \{pkt \mapsto s\}$
a10 : $pktData := pktData \cup \{pkt \mapsto 0\}$
a11 : $pktNbHops := pktNbHops \cup \{pkt \mapsto nbh\}$
a12 : $rrepLists := rrepLists \setminus \{s \mapsto rreq\}$
a13 : $netDestAddr := netDestAddr \cup \{pkt \mapsto BROADCAST\}$

end

Event *create_controlPkt* $\widehat{=}$

//(ctl) create other control packet

extends *create_controlPkt*

any

s
pkt
$fDes$
sno
nbh

where

g1 : $pkt \in PKT$
g2 : $pkt \notin ran(ndBuff \cup sentDown \cup sentUp)$
g3 : $pkt \notin (sensedPkts \cup floodedPkts)$
g4 : $s = initialSrcAddr(pkt)$
g5 : $s \in dom(floodTbl) \wedge pkt \notin floodTbl(s)$
g6 : $type(pkt) \in CONTROL$
g7 : $floodFlg(s) = TRUE$
g8 : $fDes \in ND$
g9 : $pkt \mapsto fDes \notin finalDestAddr$
g10 : $fDes \neq s$

g11 : $sno = floodSeqNo(s) + 1 \wedge s \in dom(floodSeqNo)$
g12 : $nbh \in \mathbb{Z} \wedge nbh = -1$
g13 : $s \mapsto \{pkt \mapsto nbh\} \notin nbHops$
g14 : $pkt \notin dom(pktSeqNo)$
g15 : $pkt \notin dom(pktSrc)$
g16 : $pkt \notin dom(pktFwdr)$
g17 : $pkt \notin dom(pktData)$
g18 : $pkt \notin dom(pktNbHops)$
g19 : $type(pkt) \neq RREP \wedge type(pkt) \neq RREQ$
g20 : $pkt \notin dom(netDestAddr)$
g21 : $rrepFlg(s) = FALSE$

then

a1 : $floodedPkts := floodedPkts \cup \{pkt\}$
a2 : $ndBuff := ndBuff \cup \{s \mapsto pkt\}$
a3 : $floodTbl(s) := floodTbl(s) \cup \{pkt\}$
a4 : $finalDestAddr := finalDestAddr \cup \{pkt \mapsto fDes\}$
a5 : $floodSeqNo := floodSeqNo \Leftarrow \{s \mapsto sno\}$
a6 : $nbHops := nbHops \cup \{s \mapsto \{pkt \mapsto nbh\}\}$
a7 : $pktSeqNo := pktSeqNo \Leftarrow \{pkt \mapsto sno\}$
a8 : $pktFwdr := pktFwdr \cup \{pkt \mapsto s\}$
a9 : $pktSrc := pktSrc \cup \{pkt \mapsto s\}$
a10 : $pktData := pktData \cup \{pkt \mapsto 0\}$
a11 : $pktNbHops := pktNbHops \cup \{pkt \mapsto nbh\}$
a12 : $netDestAddr := netDestAddr \cup \{pkt \mapsto BROADCAST\}$

end

Event *start_tx_dataPkt_bct* $\widehat{=}$

//(ctl) starting broadcasting a data packet to a channel

extends *start_tx_dataPkt*

any

x

pkt

nbh

where

g1 : $x \mapsto pkt \in ndBuff$
g2 : $x \mapsto pkt \notin sentDown$
g3 : $x \mapsto pkt \notin sentUp$
g4 : $x \mapsto \{pkt \mapsto nbh\} \in nbHops$
g5 : $pkt \in dom(pktFwdr)$
g6 : $pkt \in dom(pktNbHops)$
g7 : $type(pkt) = DATA$
g8 : $ran(\{x\} \lhd dom(fwdNextND)) = \varnothing$

then

a1 : $ndBuff := ndBuff \setminus \{x \mapsto pkt\}$
a2 : $sentDown := sentDown \cup \{x \mapsto pkt\}$
a3 : $pktFwdr := pktFwdr \lhd\!\!\!- \{pkt \mapsto x\}$
a4 : $pktNbHops := pktNbHops \lhd\!\!\!- \{pkt \mapsto nbh\}$
a5 : $nbHops := nbHops \setminus \{x \mapsto \{pkt \mapsto nbh\}\}$

end

Event *start_tx_dataPkt_fwd* $\widehat{=}$

//(ctl) starting unicasting a data packet to a channel via route

extends *start_tx_dataPkt*

any

x
pkt
nbh
s
nxt

where

g1 : $x \mapsto pkt \in ndBuff$
g2 : $x \mapsto pkt \notin sentDown$
g3 : $x \mapsto pkt \notin sentUp$
g4 : $x \mapsto \{pkt \mapsto nbh\} \in nbHops$
g5 : $pkt \in dom(pktFwdr)$
g6 : $pkt \in dom(pktNbHops)$
g7 : $type(pkt) = DATA$
g8 : $ran(\{x\} \lhd dom(fwdNextND)) \neq \varnothing \wedge s \in ran(\{x\} \lhd dom(fwdNextND))$
g9 : $nxt = fwdNextND(x \mapsto s)$

then

a1 : $ndBuff := ndBuff \setminus \{x \mapsto pkt\}$
a2 : $sentDown := sentDown \cup \{x \mapsto pkt\}$
a3 : $pktFwdr := pktFwdr \lhd\!\!\!- \{pkt \mapsto x\}$
a4 : $pktNbHops := pktNbHops \lhd\!\!\!- \{pkt \mapsto nbh\}$
a5 : $nbHops := nbHops \setminus \{x \mapsto \{pkt \mapsto nbh\}\}$
a6 : $netDestAddr := netDestAddr \lhd\!\!\!- \{pkt \mapsto nxt\}$

end

Event *start_tx_rreq* $\widehat{=}$

//(ctl) starting transmiting a RREQ packet to a channel

extends *start_tx_rreq*

any

x
pkt
nbh

$lsno$

where

g1 : $x \mapsto pkt \in ndBuff$

g2 : $x \mapsto pkt \notin sentDown$

g3 : $x \mapsto pkt \notin sentUp$

g4 : $x \mapsto \{pkt \mapsto nbh\} \in nbHops$

g5 : $pkt \in dom(pktFwdr)$

g6 : $pkt \in dom(pktNbHops)$

g7 : $type(pkt) = RREQ$

g8 : $x \in dom(linkSeqNo) \wedge lsno = linkSeqNo(x) + 1$

then

a1 : $ndBuff := ndBuff \setminus \{x \mapsto pkt\}$

a2 : $sentDown := sentDown \cup \{x \mapsto pkt\}$

a3 : $pktFwdr := pktFwdr \lhd\!\!\!- \{pkt \mapsto x\}$

a4 : $pktNbHops := pktNbHops \lhd\!\!\!- \{pkt \mapsto nbh\}$

a5 : $nbHops := nbHops \setminus \{x \mapsto \{pkt \mapsto nbh\}\}$

a6 : $linkSeqNo := linkSeqNo \lhd\!\!\!- \{x \mapsto lsno\}$

a7 : $netSeqNo(pkt) := lsno$

end

Event *start_tx_rrep* $\widehat{=}$

//(ctl) starting transmiting a RREP packet to a channel

extends *start_tx_controlPkt*

any

x

pkt

nbh

$lsno$

s

nxt

where

g1 : $x \mapsto pkt \in ndBuff$

g2 : $x \mapsto pkt \notin sentDown$

g3 : $x \mapsto pkt \notin sentUp$

g4 : $x \mapsto \{pkt \mapsto nbh\} \in nbHops$

g5 : $pkt \in dom(pktFwdr)$

g6 : $pkt \in dom(pktNbHops)$

g7 : $type(pkt) \neq RREQ$

g8 : $lsno \in \mathbb{N}$

g12 : $type(pkt) = RREP$

g13 : $x \in dom(rrepSeqNo) \wedge lsno = rrepSeqNo(x) + 1$

g17 : $x \mapsto s \in dom(bwdNextND) \wedge nxt = bwdNextND(x \mapsto s)$

g18 : $pkt \in dom(netDestAddr)$
g19 : $pkt \in dom(finalDestAddr) \wedge s = finalDestAddr(pkt)$

then

a1 : $ndBuff := ndBuff \setminus \{x \mapsto pkt\}$
a2 : $sentDown := sentDown \cup \{x \mapsto pkt\}$
a3 : $pktFwdr := pktFwdr \lhd\!\!\!- \{pkt \mapsto x\}$
a4 : $pktNbHops := pktNbHops \lhd\!\!\!- \{pkt \mapsto nbh\}$
a5 : $nbHops := nbHops \setminus \{x \mapsto \{pkt \mapsto nbh\}\}$
a6 : $netSeqNo(pkt) := lsno$
a7 : $rrepSeqNo := rrepSeqNo \lhd\!\!\!- \{x \mapsto lsno\}$
a8 : $netDestAddr := netDestAddr \lhd\!\!\!- \{pkt \mapsto nxt\}$

end

Event *start_tx_controlPkt* $\widehat{=}$

//(ctl) starting transmiting other control to a channel

extends *start_tx_controlPkt*

any

x
pkt
nbh
$lsno$
nxt

where

g1 : $x \mapsto pkt \in ndBuff$
g2 : $x \mapsto pkt \notin sentDown$
g3 : $x \mapsto pkt \notin sentUp$
g4 : $x \mapsto \{pkt \mapsto nbh\} \in nbHops$
g5 : $pkt \in dom(pktFwdr)$
g6 : $pkt \in dom(pktNbHops)$
g7 : $type(pkt) \neq RREQ$
g8 : $lsno \in \mathbb{N}$
g12 : $type(pkt) \neq RREP \wedge type(pkt) \neq RREQ$
g13 : $nxt \in ND$

then

a1 : $ndBuff := ndBuff \setminus \{x \mapsto pkt\}$
a2 : $sentDown := sentDown \cup \{x \mapsto pkt\}$
a3 : $pktFwdr := pktFwdr \lhd\!\!\!- \{pkt \mapsto x\}$
a4 : $pktNbHops := pktNbHops \lhd\!\!\!- \{pkt \mapsto nbh\}$
a5 : $nbHops := nbHops \setminus \{x \mapsto \{pkt \mapsto nbh\}\}$
a6 : $netSeqNo(pkt) := lsno$
a7 : $netDestAddr := netDestAddr \lhd\!\!\!- \{pkt \mapsto nxt\}$

end

Event *send_down* $\widehat{=}$

//(shd) sending a packet down to a channel

extends *send_down*

any

cn

pkt

sno

src

$data$

$fwdr$

nbh

nxt

where

g1 : $cn \mapsto pkt \in sentDown$

g2 : $cn \mapsto pkt \notin channel$

g3 : $pkt \in dom(pktSeqNo) \wedge sno = pktSeqNo(pkt)$

g4 : $pkt \in dom(pktSrc) \wedge src = pktSrc(pkt)$

g5 : $pkt \in dom(pktFwdr) \wedge fwdr = pktFwdr(pkt)$

g6 : $pkt \in dom(pktData) \wedge data = pktData(pkt)$

g7 : $pkt \in dom(pktNbHops) \wedge nbh = pktNbHops(pkt)$

g8 : $pkt \notin dom(vPktSeqNo)$

g10 : $pkt \notin dom(vPktFwdr)$

g9 : $pkt \notin dom(vPktSrc)$

g11 : $pkt \notin dom(vPktData)$

g12 : $pkt \notin dom(vPktNbHops)$

g13 : $pkt \in dom(netDestAddr) \wedge nxt = netDestAddr(pkt)$

g14 : $pkt \notin dom(envDestAddr) \wedge nxt \in ND \cup \{BROADCAST\}$

then

a1 : $channel := channel \cup \{cn \mapsto pkt\}$

a2 : $vPktSeqNo := vPktSeqNo \cup \{pkt \mapsto sno\}$

a3 : $vPktSrc := vPktSrc \cup \{pkt \mapsto src\}$

a4 : $vPktFwdr := vPktFwdr \cup \{pkt \mapsto fwdr\}$

a5 : $vPktData := vPktData \cup \{pkt \mapsto data\}$

a6 : $vPktNbHops := vPktNbHops \cup \{pkt \mapsto nbh\}$

a7 : $pktSeqNo := \{pkt\} \lhd pktSeqNo$

a8 : $pktSrc := \{pkt\} \lhd pktSrc$

a9 : $pktFwdr := \{pkt\} \lhd pktFwdr$

a10 : $pktData := \{pkt\} \lhd pktData$

a11 : $pktNbHops := \{pkt\} \lhd pktNbHops$

a12 : $envDestAddr := envDestAddr \cup \{pkt \mapsto nxt\}$

end

Event *find_neighbours* $\widehat{=}$

//(env) channel finds the neighbour list of the forwarder

extends *find_neighbours*

any

s

nbs

f

pkt

nxt

where

g1 : $channel \neq \varnothing$

g2 : $pkt \in ran(channel)$

g3 : $s = initialSrcAddr(pkt)$

g4 : $f \mapsto pkt \in channel$

g5 : $nbs \subseteq ND \wedge nbs = wsnLinks[\{f\}] \wedge nbs \neq \varnothing$

g6 : $envNeighbours = \varnothing$

g7 : $pkt \in dom(netDestAddr) \wedge nxt = netDestAddr(pkt)$

g8 : $nxt = BROADCAST$

then

a1 : $envNeighbours := envNeighbours \cup (\{pkt\} \times nbs)$

end

Event *assign_forwarder* $\widehat{=}$

//(env) channel assigns specific forwarder

extends *assign_forwarder*

any

s

nb

pkt

f

nbs

nxt

where

g1 : $channel \neq \varnothing$

g2 : $pkt \in ran(channel)$

g3 : $s = initialSrcAddr(pkt)$

g4 : $f \mapsto pkt \in channel$

g5 : $nbs = wsnLinks[\{f\}] \wedge nbs \neq \varnothing$

g6 : $nb \in nbs$

g7 : $pkt \in dom(netDestAddr) \wedge nxt = netDestAddr(pkt)$

g8 : $nxt \neq BROADCAST$

g9 : $nb = nxt$

then

a1 : $envNeighbours := envNeighbours \cup \{pkt \mapsto nb\}$

end

Event *send_up* $\widehat{=}$

//(env) channel sends the neighbour information

extends *send_up*

any

pkt
$nbrs$
f
sno
src
$fwdr$
$data$
nbh
nxt

where

g1 : $pkt \in ran(sentDown)$
g2 : $f \mapsto pkt \in sentDown$
g3 : $f \mapsto pkt \notin sentUp$
g4 : $f \mapsto pkt \in channel$
g5 : $nbrs \in \{n | n \in (\mathbb{P}(ND)) \cup \{\{FAILED\_XMIT\}\}\}$
g6 : $nbrs \neq \varnothing$
g7 : $ran(\{pkt\} \lhd envNeighbours) \in (\mathbb{P}(ND) \cup \{\{FAILED\_XMIT\}\})$
g8 : $nbrs = ran(\{pkt\} \lhd envNeighbours) \wedge nbrs \neq \varnothing$
g9 : $pkt \notin dom(ctlNeighbours)$
g10 : $pkt \in dom(vPktSeqNo) \wedge sno = vPktSeqNo(pkt)$
g11 : $pkt \in dom(vPktSrc) \wedge src = vPktSrc(pkt)$
g12 : $pkt \in dom(vPktFwdr) \wedge fwdr = vPktFwdr(pkt)$
g13 : $pkt \in dom(vPktData) \wedge data = vPktData(pkt)$
g14 : $pkt \in dom(vPktNbHops) \wedge nbh = vPktNbHops(pkt)$
g15 : $pkt \notin dom(pktSeqNo)$
g16 : $pkt \notin dom(pktSrc)$
g17 : $pkt \notin dom(pktFwdr)$
g18 : $pkt \notin dom(pktData)$
g19 : $pkt \notin dom(pktNbHops)$
g20 : $pkt \mapsto nxt \in envDestAddr$

then

a1 : $ctlNeighbours := ctlNeighbours \cup (\{pkt\} \times nbrs)$
a2 : $envNeighbours := \varnothing$
a3 : $channel := channel \setminus \{f \mapsto pkt\}$

a4 : $sentDown := sentDown \setminus \{f \mapsto pkt\}$
a5 : $sentUp := sentUp \cup \{f \mapsto pkt\}$
a6 : $pktSeqNo := pktSeqNo \cup \{pkt \mapsto sno\}$
a7 : $pktSrc := pktSrc \cup \{pkt \mapsto src\}$
a8 : $pktFwdr := pktFwdr \cup \{pkt \mapsto fwdr\}$
a9 : $pktData := pktData \cup \{pkt \mapsto data\}$
a10 : $pktNbHops := pktNbHops \cup \{pkt \mapsto nbh\}$
a11 : $vPktSeqNo := \{pkt\} \lhd vPktSeqNo$
a12 : $vPktSrc := \{pkt\} \lhd vPktSrc$
a13 : $vPktFwdr := \{pkt\} \lhd vPktFwdr$
a14 : $vPktData := \{pkt\} \lhd vPktData$
a15 : $vPktNbHops := \{pkt\} \lhd vPktNbHops$
a16 : $envDestAddr := envDestAddr \setminus \{pkt \mapsto nxt\}$

end

Event *receive_dataPkt* $\widehat{=}$

//(ctl) each neighbour node receives a data packet from channel

extends *receive_dataPkt*

any

s
nb
pkt
nbh
f

where

g1 : $pkt \in ran(sentUp \cup sentDown)$
g2 : $pkt \mapsto nb \in ctlNeighbours$
g3 : $nb \mapsto pkt \notin ndBuff$
g4 : $nb \notin dom(sentUp \cup sentDown)$
g5 : $s = initialSrcAddr(pkt) \wedge s \neq nb$
g6 : $pkt \in dom(finalDestAddr) \wedge nb \neq finalDestAddr(pkt)$
g7 : $nb \in dom(floodTbl) \wedge pkt \notin floodTbl(nb)$
g8 : $pkt \in dom(pktNbHops) \wedge nbh = pktNbHops(pkt) + 1$
g9 : $pkt \mapsto f \in pktFwdr$
g10 : $type(pkt) = DATA$

then

a1 : $ndBuff := ndBuff \cup \{nb \mapsto pkt\}$
a2 : $ctlNeighbours := ctlNeighbours \setminus \{pkt \mapsto nb\}$
a3 : $floodTbl(nb) := floodTbl(nb) \cup \{pkt\}$
a4 : $nbHops := nbHops \cup \{nb \mapsto \{pkt \mapsto nbh\}\}$

end

Event *receive_dup_dataPkt* $\widehat{=}$

//(ctl) each neighbour node including sink receives a duplicated data packet from channel

extends *receive_dup_dataPkt*

any

nb

pkt

f

where

g1 : $pkt \in dom(ctlNeighbours)$

g2 : $pkt \mapsto nb \in ctlNeighbours$

g3 : $nb \in dom(floodTbl) \wedge pkt \in floodTbl(nb)$

g4 : $pkt \mapsto f \in pktFwdr$

g5 : $type(pkt) = DATA$

then

a1 : $ctlNeighbours := ctlNeighbours \setminus \{pkt \mapsto nb\}$

end

Event *receive_rreqPkt* $\widehat{=}$

//(ctl) each neighbour node receives a RREQ packet from channel

extends *receive_controlPkt*

any

s

nb

pkt

nbh

f

where

g1 : $pkt \in ran(sentUp \cup sentDown)$

g2 : $pkt \mapsto nb \in ctlNeighbours$

g3 : $nb \mapsto pkt \notin ndBuff$

g4 : $nb \notin dom(sentUp \cup sentDown)$

g5 : $s = initialSrcAddr(pkt) \wedge s \neq nb$

g6 : $pkt \in dom(finalDestAddr) \wedge nb \neq finalDestAddr(pkt)$

g7 : $nb \in dom(floodTbl) \wedge pkt \notin floodTbl(nb)$

g8 : $pkt \in dom(pktNbHops) \wedge nbh = pktNbHops(pkt) + 1$

g9 : $pkt \mapsto f \in pktFwdr$

g10 : $type(pkt) \in CONTROL$

g11 : $f \in ND \wedge nb \in ND$

g12 : $f \mapsto nb \notin updateNbrs$

g13 : $type(pkt) = RREQ$

then

a1 : $ndBuff := ndBuff \cup \{nb \mapsto pkt\}$
a2 : $ctlNeighbours := ctlNeighbours \setminus \{pkt \mapsto nb\}$
a3 : $floodTbl(nb) := floodTbl(nb) \cup \{pkt\}$
a4 : $nbHops := nbHops \cup \{nb \mapsto \{pkt \mapsto nbh\}\}$
a5 : $updateNbrs := updateNbrs \cup \{f \mapsto nb\}$

end

Event *receive_dup_rreqPkt* $\widehat{=}$

//(ctl) each neighbour node including sink receives a duplicated RREQ packet from channel

extends *receive_dup_controlPkt*

any

nb
pkt
f

where

g1 : $pkt \in dom(ctlNeighbours)$
g2 : $pkt \mapsto nb \in ctlNeighbours$
g3 : $nb \in dom(floodTbl) \wedge pkt \in floodTbl(nb)$
g4 : $pkt \mapsto f \in pktFwdr$
g5 : $type(pkt) \neq DATA$
g6 : $f \in ND \wedge nb \in ND$
g7 : $f \mapsto nb \notin updateNbrs$
g8 : $type(pkt) = RREQ$

then

a1 : $ctlNeighbours := ctlNeighbours \setminus \{pkt \mapsto nb\}$

end

Event *receive_rrepPkt* $\widehat{=}$

//(ctl) each neighbour node receives a RREP packet from channel

extends *receive_controlPkt*

any

s
nb
pkt
nbh
f

where

g1 : $pkt \in ran(sentUp \cup sentDown)$
g2 : $pkt \mapsto nb \in ctlNeighbours$
g3 : $nb \mapsto pkt \notin ndBuff$
g4 : $nb \notin dom(sentUp \cup sentDown)$
g5 : $s = initialSrcAddr(pkt) \wedge s \neq nb$

g6 : $pkt \in dom(finalDestAddr) \wedge nb \neq finalDestAddr(pkt)$

g7 : $nb \in dom(floodTbl) \wedge pkt \notin floodTbl(nb)$

g8 : $pkt \in dom(pktNbHops) \wedge nbh = pktNbHops(pkt) + 1$

g9 : $pkt \mapsto f \in pktFwdr$

g10 : $type(pkt) \in CONTROL$

g11 : $f \in ND \wedge nb \in ND$

g12 : $f \mapsto nb \notin updateNbrs$
//nb has not considered to update in the route table yet

g13 : $type(pkt) = RREP$

then

a1 : $ndBuff := ndBuff \cup \{nb \mapsto pkt\}$

a2 : $ctlNeighbours := ctlNeighbours \setminus \{pkt \mapsto nb\}$

a3 : $floodTbl(nb) := floodTbl(nb) \cup \{pkt\}$

a4 : $nbHops := nbHops \cup \{nb \mapsto \{pkt \mapsto nbh\}\}$

a5 : $updateNbrs := updateNbrs \cup \{f \mapsto nb\}$

//put nb into updateNbrs for update it into the route table later

end

Event *receive_dup_rrepPkt* $\widehat{=}$

//(ctl) each neighbour node including sink receives a duplicated RREP packet from channel

extends *receive_dup_controlPkt*

any

nb

pkt

f

where

g1 : $pkt \in dom(ctlNeighbours)$

g2 : $pkt \mapsto nb \in ctlNeighbours$

g3 : $nb \in dom(floodTbl) \wedge pkt \in floodTbl(nb)$

g4 : $pkt \mapsto f \in pktFwdr$

g5 : $type(pkt) \neq DATA$

g6 : $f \in ND \wedge nb \in ND$

g7 : $f \mapsto nb \notin updateNbrs$

//nb has not considered to update in the route table yet

g8 : $type(pkt) = RREP$

then

a1 : $ctlNeighbours := ctlNeighbours \setminus \{pkt \mapsto nb\}$

end

Event *receive_controlPkt* $\widehat{=}$

//(ctl) each neighbour node receives a flood packet from channel

extends *receive_controlPkt*

any

s

nb

pkt

nbh

f

where

g1 : $pkt \in ran(sentUp \cup sentDown)$

g2 : $pkt \mapsto nb \in ctlNeighbours$

g3 : $nb \mapsto pkt \notin ndBuff$

g3 : $nb \notin dom(sentUp \cup sentDown)$

g4 : $s = initialSrcAddr(pkt) \wedge s \neq nb$

g5 : $pkt \in dom(finalDestAddr) \wedge nb \neq finalDestAddr(pkt)$

g6 : $nb \in dom(floodTbl) \wedge pkt \notin floodTbl(nb)$

g8 : $pkt \in dom(pktNbHops) \wedge nbh = pktNbHops(pkt) + 1$

g9 : $pkt \mapsto f \in pktFwdr$

g10 : $type(pkt) \in CONTROL$

g11 : $f \in ND \wedge nb \in ND$

g12 : $f \mapsto nb \notin updateNbrs$

//nb has not considered to update in the route table yet

g13 : $type(pkt) \neq RREQ \wedge type(pkt) \neq RREP$

then

a1 : $ndBuff := ndBuff \cup \{nb \mapsto pkt\}$

a2 : $ctlNeighbours := ctlNeighbours \setminus \{pkt \mapsto nb\}$

a3 : $floodTbl(nb) := floodTbl(nb) \cup \{pkt\}$

a4 : $nbHops := nbHops \cup \{nb \mapsto \{pkt \mapsto nbh\}\}$

a5 : $updateNbrs := updateNbrs \cup \{f \mapsto nb\}$

//put nb into updateNbrs for update it into the route table later

end

Event *receive_dup_controlPkt* $\widehat{=}$

//(ctl) each neighbour node including sink receives a duplicated flood packet from channel

extends *receive_dup_controlPkt*

any

nb

pkt

f

where

g1 : $pkt \in dom(ctlNeighbours)$

g2 : $pkt \mapsto nb \in ctlNeighbours$

g3 : $nb \in dom(floodTbl) \wedge pkt \in floodTbl(nb)$ //is a duplicted packet

g4 : $pkt \mapsto f \in pktFwdr$

g5 : $type(pkt) \neq DATA$

g6 : $f \in ND \wedge nb \in ND$

g7 : $f \mapsto nb \notin updateNbrs$

//nb has not considered to update in the route table yet

g8 : $type(pkt) \neq RREQ \wedge type(pkt) \neq RREP$

then

a1 : $ctlNeighbours := ctlNeighbours \setminus \{pkt \mapsto nb\}$

end

Event *dest_recv_dataPkt* $\widehat{=}$

//(ctl) destination receives a data packet

extends *dest_recv_dataPkt*

any

s

des

pkt

$data$

f

where

g1 : $pkt \in PKT$

g2 : $pkt \in dom(finalDestAddr) \wedge des = finalDestAddr(pkt)$

g3 : $des \in dom(destBuff)$

g4 : $pkt \in ran(sentUp \cup sentDown) \setminus destBuff(des)$

g5 : $s = initialSrcAddr(pkt)$

g6 : $\{pkt\} \lhd ctlNeighbours \neq \varnothing$

g7 : $ran(\{pkt\} \lhd ctlNeighbours) \neq \{FAILED\_XMIT\}$

g8 : $des \in (ran(\{pkt\} \lhd ctlNeighbours))$

g9 : $des \in dom(recvedData)$

g10 : $pkt \in dom(pktData) \wedge data = pktData(pkt)$

g11 : $type(pkt) = DATA$

g12 : $pkt \mapsto f \in pktFwdr$

then

a1 : $destBuff(des) := destBuff(des) \cup \{pkt\}$

a2 : $ctlNeighbours := ctlNeighbours \setminus \{pkt \mapsto des\}$

a3 : $floodTbl(des) := floodTbl(des) \cup \{pkt\}$

a4 : $recvedData(des) := recvedData(des) \cup \{data\}$

end

Event *dest_recv_rreqPkt* $\widehat{=}$

//(ctl) destination receives a RREQ packet

extends *dest_recv_controlPkt*

any

s

des

pkt

f

where

g1 : $pkt \in PKT$

g2 : $pkt \in dom(finalDestAddr) \wedge des = finalDestAddr(pkt)$

g3 : $des \in dom(destBuff)$

g4 : $pkt \in ran(sentUp \cup sentDown) \setminus destBuff(des)$

g5 : $s = initialSrcAddr(pkt)$

g6 : $\{pkt\} \lhd ctlNeighbours \neq \varnothing$

g7 : $ran(\{pkt\} \lhd ctlNeighbours) \neq \{FAILED\_XMIT\}$

g8 : $des \in (ran(\{pkt\} \lhd ctlNeighbours))$

g9 : $type(pkt) \in CONTROL$

g10 : $pkt \mapsto f \in pktFwdr$

g11 : $type(pkt) \neq DATA$

g12 : $f \in ND \wedge des \in ND$

g13 : $f \mapsto des \notin updateNbrs$

g14 : $type(pkt) = RREQ$

g15 : $des \mapsto pkt \notin rrepLists$

then

a1 : $destBuff(des) := destBuff(des) \cup \{pkt\}$

a2 : $ctlNeighbours := ctlNeighbours \setminus \{pkt \mapsto des\}$

a3 : $floodTbl(des) := floodTbl(des) \cup \{pkt\}$

a4 : $updateNbrs := updateNbrs \cup \{f \mapsto des\}$

a5 : $rrepLists := rrepLists \cup \{des \mapsto pkt\}$

end

Event *dest_recv_rrepPkt* $\widehat{=}$

//(ctl) destination receives a RREQ packet

extends *dest_recv_controlPkt*

any

s

des

pkt

f

where

g1 : $pkt \in PKT$

g2 : $pkt \in dom(finalDestAddr) \wedge des = finalDestAddr(pkt)$

g3 : $des \in dom(destBuff)$

g4 : $pkt \in ran(sentUp \cup sentDown) \setminus destBuff(des)$

g5 : $s = initialSrcAddr(pkt)$

g6 : $\{pkt\} \lhd ctlNeighbours \neq \varnothing$

g7 : $ran(\{pkt\} \lhd ctlNeighbours) \neq \{FAILED\_XMIT\}$
g8 : $des \in (ran(\{pkt\} \lhd ctlNeighbours))$
g9 : $type(pkt) \in CONTROL$
g10 : $pkt \mapsto f \in pktFwdr$
g11 : $type(pkt) \neq DATA$
g12 : $f \in ND \wedge des \in ND$
g13 : $f \mapsto des \notin updateNbrs$
g14 : $type(pkt) = RREP$

then

a1 : $destBuff(des) := destBuff(des) \cup \{pkt\}$
a2 : $ctlNeighbours := ctlNeighbours \setminus \{pkt \mapsto des\}$
a3 : $floodTbl(des) := floodTbl(des) \cup \{pkt\}$
a4 : $updateNbrs := updateNbrs \cup \{f \mapsto des\}$

end

Event *dest_recv_controlPkt* $\widehat{=}$

//(ctl) sink receives a flood packet

extends *dest_recv_controlPkt*

any

s
des
pkt
f

where

g1 : $pkt \in PKT$
g2 : $pkt \in dom(finalDestAddr) \wedge des = finalDestAddr(pkt)$
g3 : $des \in dom(destBuff)$
g4 : $pkt \in ran(sentUp \cup sentDown) \setminus destBuff(des)$
g5 : $s = initialSrcAddr(pkt)$
g6 : $\{pkt\} \lhd ctlNeighbours \neq \varnothing$
g7 : $ran(\{pkt\} \lhd ctlNeighbours) \neq \{FAILED\_XMIT\}$
g8 : $des \in (ran(\{pkt\} \lhd ctlNeighbours))$
g9 : $type(pkt) \in CONTROL$
g10 : $pkt \mapsto f \in pktFwdr$
g11 : $type(pkt) \neq DATA$
g12 : $f \in ND \wedge des \in ND$
g13 : $f \mapsto des \notin updateNbrs$
g14 : $type(pkt) \neq RREQ \wedge type(pkt) \neq RREP$

then

a1 : $destBuff(des) := destBuff(des) \cup \{pkt\}$
a2 : $ctlNeighbours := ctlNeighbours \setminus \{pkt \mapsto des\}$
a3 : $floodTbl(des) := floodTbl(des) \cup \{pkt\}$

a4 : $updateNbrs := updateNbrs \cup \{f \mapsto des\}$

end

Event *actuating* $\widehat{=}$

//(ctl) alerting emergency: sensed data is above safety threshold

extends *actuating*

any

act

$data$

where

g1 : $act \in Actuators$

g2 : $act \in dom(recvedData)$

g3 : $data \in recvedData(act)$

g4 : $data \geq safetyThreshold$

g5 : $emergencyAlert(act) = FALSE$

then

a1 : $recvedData(act) := recvedData(act) \setminus \{data\}$

a2 : $emergencyAlert(act) := TRUE$

end

Event *no_actuating* $\widehat{=}$

//(ctl) not alerting emergency sensed data is below safety threshold

extends *no_actuating*

any

act

$data$

where

g1 : $act \in Actuators$

g2 : $act \in dom(recvedData)$

g3 : $data \in recvedData(act)$

g4 : $data < safetyThreshold$

then

a1 : $recvedData(act) := recvedData(act) \setminus \{data\}$

end

Event *reset_actuatingStatus* $\widehat{=}$

//(ctl) resetting emergency alert status

extends *reset_actuatingStatus*

any

act

where

g1 : $act \in dom(emergencyAlert)$

g2 : $emergencyAlert(act) = TRUE$

then

a1 : $emergencyAlert(act) := FALSE$

end

Event *add_bwdRouteEntry* $\widehat{=}$

//(ctl) add new entry into the backwards route table

extends *add_routeEntry*

any

x

y

s

pkt

sNo

$hCnt$

where

g1 : $updateNbrs \neq \varnothing$

//there is some nodes that have to update thier backwards route table

g2 : $x \in ND \wedge y \in ND \wedge x \mapsto y \in updateNbrs$

g3 : $pkt \in dom(initialSrcAddr) \wedge s = initialSrcAddr(pkt)$

g4 : $y \mapsto s \notin bwdRouteTbl$

//in case these new pairs don't appear in the backwards route table

g5 : $pkt \mapsto x \in pktFwdr$

g6 : $sNo = netSeqNo(pkt) \wedge y \mapsto x \mapsto sNo \notin bwdSeqNo$

g7 : $hCnt \in \mathbb{N} \wedge pkt \in dom(pktNbHops) \wedge hCnt = pktNbHops(pkt) + 1$
$\wedge\, y \mapsto x \mapsto hCnt \notin bwdHopCnt$

g8 : $ctlNeighbours = \varnothing$

g9 : $type(pkt) = RREQ$

g10 : $type(pkt) = RREQ$

g12 : $pkt \in ran(sentUp)$

then

a1 : $bwdRouteTbl := bwdRouteTbl \cup \{y \mapsto s\}$

a2 : $bwdNextND := bwdNextND \cup \{y \mapsto s \mapsto x\}$

a3 : $bwdSeqNo := bwdSeqNo \cup \{y \mapsto s \mapsto sNo\}$

a4 : $bwdHopCnt := bwdHopCnt \cup \{y \mapsto s \mapsto hCnt\}$

a5 : $updateNbrs := updateNbrs \setminus \{x \mapsto y\}$

end

Event *add_fwdrouteEntry* $\widehat{=}$

// add new entry into the forwards route table

extends *add_routeEntry2*

any

x

y

s

pkt

sNo

$hCnt$

where

g1 : $updateNbrs \neq \varnothing$

//there is some nodes that have to update thier forwards routing table

g2 : $x \in ND \wedge y \in ND \wedge x \mapsto y \in updateNbrs$

g3 : $pkt \in dom(initialSrcAddr) \wedge s = initialSrcAddr(pkt)$

// in case these new pairs don't appear in the forwards routing table

g4 : $pkt \mapsto x \in pktFwdr$

g5 : $sNo = netSeqNo(pkt) \wedge y \mapsto x \mapsto sNo \notin bwdSeqNo$

g6 : $hCnt \in \mathbb{N} \wedge pkt \in dom(pktNbHops) \wedge hCnt = pktNbHops(pkt) + 1$
$\wedge\, y \mapsto x \mapsto hCn \notin bwdHopCnt$

g7 : $ctlNeighbours = \varnothing$

g8 : $type(pkt) \neq DATA \wedge type(pkt) \neq RREQ$

g9 : $y \mapsto s \notin fwdRouteTbl$

g10 : $type(pkt) = RREP$

g11 : $pkt \in ran(sentUp)$

then

a1 : $updateNbrs := updateNbrs \setminus \{x \mapsto y\}$

a2 : $fwdRouteTbl := fwdRouteTbl \cup \{y \mapsto s\}$

a3 : $fwdNextND := fwdNextND \cup \{y \mapsto s \mapsto x\}$

a4 : $fwdSeqNo := fwdSeqNo \cup \{y \mapsto s \mapsto sNo\}$

a5 : $fwdHopCnt := fwdHopCnt \cup \{y \mapsto s \mapsto hCnt\}$

end

Event *add_routeEntry3* $\widehat{=}$

//(ctl) add new entry into the route table

extends *add_routeEntry3*

any

x

y

s

pkt

sNo

$hCnt$

where

g1 : $updateNbrs \neq \varnothing$

//there is some nodes that have to update thier route table

g2 : $x \in ND \wedge y \in ND \wedge x \mapsto y \in updateNbrs$

g3 : $pkt \in dom(initialSrcAddr) \wedge s = initialSrcAddr(pkt)$

// in case these new pairs don't appear in the route table

g4 : $pkt \mapsto x \in pktFwdr$

g5 : $sNo = netSeqNo(pkt) \wedge y \mapsto x \mapsto sNo \notin bwdSeqNo$

g7 : $hCnt \in \mathbb{N} \wedge pkt \in dom(pktNbHops) \wedge hCnt = pktNbHops(pkt) + 1$
$\wedge\ y \mapsto x \mapsto hCnt \notin bwdHopCnt$
g8 : $ctlNeighbours = \varnothing$
g9 : $type(pkt) \neq DATA \wedge type(pkt) \neq RREQ$
g10 : $type(pkt) \neq RREQ \wedge type(pkt) \neq RREP$

then

a1 : $updateNbrs := updateNbrs \setminus \{x \mapsto y\}$

end

Event *finish_tx_pkt* $\widehat{=}$

// (ctl) finishing packet transmission

extends *finish_tx_pkt*

any

pkt
f

where

g1 : $\{pkt\} \lhd ctlNeighbours = \varnothing$
g2 : $f \in ND \wedge f \mapsto pkt \in sentUp$
g3 : $pkt \in ran(sentUp) \wedge pkt \notin ran(sentDown) \wedge pkt \in ran(ndBuff)$

then

a1 : $sentUp := sentUp \setminus \{f \mapsto pkt\}$

end

Event *final_tx_pkt* $\widehat{=}$

//(ctl) the transmitted packet is removed from middleware indicating completion of transmission mechanism

extends *final_tx_pkt*

any

pkt
sk

where

g1 : $pkt \in PKT$
g2 : $pkt \in ran(sentUp) \wedge pkt \notin ran(sentDown) \wedge pkt \notin ran(ndBuff)$
g3 : $pkt \in dom(finalDestAddr) \wedge sk = finalDestAddr(pkt)$
g4 : $sk \in ND$
g5 : $pkt \in destBuff(sk)$
g6 : $\{pkt\} \lhd ctlNeighbours = \varnothing$

then

a1 : $sentUp := sentUp \rhd \{pkt\}$

end

Event *lose_all_neighbours* $\widehat{=}$

//(env) channel loses all neighbour

extends *lose_all_neighbours*

any

s

nbs

f

pkt

where

g1 : $channel \neq \varnothing$

g2 : $pkt \in ran(channel)$

g3 : $s = initialSrcAddr(pkt)$

g4 : $f \mapsto pkt \in channel$

g5 : $nbs = wsnLinks[\{f\}] \wedge nbs = \varnothing$

then

a1 : $envNeighbours := envNeighbours \cup \{pkt \mapsto FAILED\_XMIT\}$

end

Event *lose_specifc_neighbour* $\widehat{=}$

//(env) channel loses a specific neighbour

extends *lose_specifc_neighbour*

any

s

nbs

f

pkt

where

g1 : $channel \neq \varnothing$

g2 : $pkt \in ran(channel)$

g3 : $s = initialSrcAddr(pkt)$

g4 : $f \mapsto pkt \in channel$

g5 : $nbs = wsnLinks[\{f\}] \wedge nbs \neq \varnothing$

then

a1 : $envNeighbours := envNeighbours \cup \{pkt \mapsto FAILED\_XMIT\}$

end

Event *lose_pkt* $\widehat{=}$

//(ctl) each node records a lost packet

extends *lose_pkt*

any

f

pkt

where

g1 : $pkt \in ran(sentUp) \wedge pkt \notin ran(sentDown)$

g2 : $pkt \notin ran(recLostPkts)$

g3 : $pkt \notin ran(ndBuff)$

g4 : $f \mapsto pkt \in sentUp \wedge f \mapsto pkt \notin sentDown$
g5 : $f \mapsto pkt \notin recLostPkts$
g6 : $ran(\{pkt\} \lhd ctlNeighbours) = \{FAILED\_XMIT\}$
then

a1 : $sentUp := sentUp \rhd \{pkt\}$
a2 : $recLostPkts := recLostPkts \cup \{f \mapsto pkt\}$
a3 : $ctlNeighbours := ctlNeighbours \setminus \{pkt \mapsto FAILED\_XMIT\}$
end

END

C.7 Sixth refinement:Data packet and route error (RRER) mechanism

C.7.1 Machine:M6

MACHINE M6
REFINES M5
SEES C4
VARIABLES

$destBuff$ //(ctl) nodes' receiving buffer
$finalDestAddr$ //(ctl) final destination address
$ndBuff$ //(ctl) nodes' waiting buffer
$sensingNDs$ //(ctl) sensing nodes
$sensedPkts$ //(ctl) sensed packets
$floodedPkts$ //(ctl) flooded packets
$floodTbl$ //(ctl) historical received packets used for detect duplicated packets
$ctlNeighbours$ //(ctl) neighbour lists
$recLostPkts$ //(ctl) record recLostPkts packet at each node
$floodFlg$ //(ctl) flood flag
$envSensedFlg$ //(env) sensing period
$envData$ //(env) environmental/conditional data
$ctlSensedFlg$ //(ctl) sensing period
$senseBuff$ //(ctl) sense buffer
$recvedData$ //(ctl) received data at destination
$emergencyAlert$ //(ctl) actuating status
$pktData$ //(ctl) data attached in transmitted packet
$sentDown$ //(ctl) transmitted packet to channel

$sentUp$ //(ctl) transmitted packet up from channel

$channel$ //(env) channel

$envNeighbours$ //(env) neighbour list

$wsnLinks$ //(env) dynamic WSN links

$crashedLinks$ //(env) crashed WSN Links

$dataSeqNo$

//(ctl) local nodes' current data sequence number at application layer
$floodSeqNo$

//(ctl) local nodes' current broadcast sequence number at application layer
$nbHops$ //(ctl) neighbour's hop count

$pktSrc$ //(ctl) src address attached in transmitted packet

$pktSeqNo$ //(ctl) sequence number attached in transmitted packet

$pktFwdr$ //(ctl) fowarder attached in transmitted packet

$pktNbHops$ //(ctl) hop count attached in transmitted packet

$vPktSeqNo$ //(env) virtual packet at channel

$vPktSrc$ //(env) virtual packet at channel

$vPktFwdr$ //(env) virtual packet at channel

$vPktData$ //(env) virtual packet at channel

$vPktNbHops$ //(env) virtual packet at chennel

$bwdRouteTbl$ //(ctl) backwards route structure

$bwdNextND$ //(ctl) backwards route table: field next node

$bwdSeqNo$ //(ctl) backwards route table: field sequence number

$bwdHopCnt$ //(ctl) backwards route table: field hop count

$netSeqNo$ //(ctl) link sequence number in a packet

$linkSeqNo$ //(ctl) link sequence number of each node

$updateNbrs$ //(ctl) for update neighbours in the neighbour table

$fwdRouteTbl$ //(ctl) forwards route table structure

$fwdNextND$ //(ctl) forwards route table: field next node

$fwdSeqNo$ //(ctl) forwards route table: field sequence number

$fwdHopCnt$ //(ctl) forwards route table: field hop count

$rrepLists$ //(ctl) route reply lists

$rrepSeqNo$ //(ctl) sequence of route reply at each node

$netDestAddr$ //(ctl) destination address of RREP

$envDestAddr$ //(env) destination addr for channel

$rrepFlg$ //(ctl) RREP flag

$rrerLists$ //(ctl) route error lists

$pktErrND$ //(ctl) fault node attached in transmitted packet

$errND$ //(ctl) fault node recorded at each node

$rrerFlg$ //(ctl) RRER flag

INVARIANTS

inv6_1 : $rrerLists \in ND \leftrightarrow PKT$

inv6_2 : $pktErrND \in PKT \nrightarrow ND$

inv6_4 : $errND \in ND \leftrightarrow ND$

inv6_5 : $rrerFlg \in ND \rightarrow BOOL$

saf6_1 : $errND \subseteq fwdRouteTbl$

EVENTS

Initialisation

extended

begin

int0_3 : $destBuff := ND \times \{\varnothing\}$

int0_4 : $finalDestAddr := \varnothing$

int1_1 : $ndBuff := \varnothing$

int1_3 : $sensingNDs := \varnothing$

int1_4 : $sensedPkts := \varnothing$

int1_6 : $floodedPkts := \varnothing$

int1_7 : $floodTbl := ND \times \{\varnothing\}$

int1_8 : $ctlNeighbours := \varnothing$

int1_12 : $floodFlg := ND \times \{FALSE\}$

inv1_20 : $recLostPkts := \varnothing$

int2_1 : $envSensedFlg := (ND \setminus Destination) \times \{FALSE\}$

int2_2 : $envData := (ND \setminus Destination) \times \{0\}$

int2_3 : $ctlSensedFlg := (ND \setminus Destination) \times \{FALSE\}$

int2_4 : $senseBuff := (ND \setminus Destination) \times \{\varnothing\}$

int2_5 : $recvedData := Destination \times \{\varnothing\}$

int2_6 : $emergencyAlert := Actuators \times \{FALSE\}$

int2_7 : $pktData := \varnothing$

int3_1 : $sentUp := \varnothing$

int3_2 : $sentDown := \varnothing$

int3_3 : $channel := \varnothing$

int3_4 : $envNeighbours := \varnothing$

int3_5 : $wsnLinks := \varnothing$

int3_6 : $crashedLinks := \varnothing$

int3_7 : $dataSeqNo := ND \times \{0\}$

int3_8 : $floodSeqNo := ND \times \{0\}$

int3_9 : $nbHops := \varnothing$

int3_10 : $pktSeqNo := \varnothing$

int3_11 : $pktSrc := \varnothing$

int3_12 : $pktFwdr := \varnothing$

int3_13 : $pktNbHops := \varnothing$
int3_14 : $vPktSeqNo := \varnothing$
int3_15 : $vPktSrc := \varnothing$
int3_16 : $vPktFwdr := \varnothing$
int2_8 : $vPktData := \varnothing$
int3_18 : $vPktNbHops := \varnothing$
int4_1 : $bwdRouteTbl := \varnothing$
int4_2 : $bwdSeqNo := \varnothing$
int4_3 : $bwdNextND := \varnothing$
int4_4 : $bwdHopCnt := \varnothing$
int4_5 : $netSeqNo := PKT \times \{0\}$
int4_6 : $linkSeqNo := ND \times \{0\}$
int4_7 : $updateNbrs := \varnothing$
int5_1 : $fwdRouteTbl := \varnothing$
int5_2 : $fwdSeqNo := \varnothing$
int5_3 : $fwdNextND := \varnothing$
int5_4 : $fwdHopCnt := \varnothing$
int5_5 : $rrepLists := \varnothing$
int5_6 : $rrepSeqNo := ND \times \{0\}$
int5_7 : $netDestAddr := \varnothing$
int5_8 : $envDestAddr := \varnothing$
int5_9 : $rrepFlg := ND \times \{FALSE\}$
int6_1 : $rrerLists := \varnothing$
int6_2 : $pktErrND := \varnothing$
int6_4 : $errND := \varnothing$
int6_5 : $rrerFlg := ND \times \{FALSE\}$

end

Event *add_link* $\widehat{=}$

//(env) channel sets a topology

extends *add_link*

any

x

y

where

g1 : $x \in ND \land y \in ND$
g2 : $x \mapsto y \notin wsnLinks$
g3 : $x \neq y$

then

a1 : $wsnLinks := wsnLinks \cup \{x \mapsto y\}$

end

Event *disconnect_link* $\widehat{=}$

//(env) disconnecting a link

extends *disconnect_link*

any

x

y

where

g1 : $x \in ND \wedge y \in ND$

g2 : $x \mapsto y \in wsnLinks$

g3 : $x \mapsto y \notin crashedLinks$

then

a1 : $crashedLinks := crashedLinks \cup \{x \mapsto y\}$

a2 : $wsnLinks := wsnLinks \setminus \{x \mapsto y\}$

end

Event *recover_link* $\widehat{=}$

//(env) recovering a link

extends *recover_link*

any

x

y

where

g1 : $x \in ND \wedge y \in ND$

g2 : $x \mapsto y \in crashedLinks$

g3 : $x \mapsto y \notin wsnLinks$

then

a1 : $wsnLinks := wsnLinks \cup \{x \mapsto y\}$

a2 : $crashedLinks := crashedLinks \setminus \{x \mapsto y\}$

end

Event *start_sensing* $\widehat{=}$

//(env) activating each node to sense data

extends *start_sensing*

any

x

sd

where

g1 : $x \in ND \setminus Destination$

g2 : $envSensedFlg(x) = FALSE$

g3 : $sd \in randomFn$

then

a1 : $envSensedFlg(x) := TRUE$

a2 : $envData(x) := sd$

end

Event *sensing* $\widehat{=}$

//(shd) each node sense a data from environment

extends *sensing*

any

x

sf

sd

where

g1 : $x \in ND \setminus Destination$

g2 : $sf = envSensedFlg(x)$

g3 : $envSensedFlg(x) = TRUE$

g4 : $ctlSensedFlg(x) = FALSE$

g5 : $sd = envData(x)$

then

a1 : $ctlSensedFlg(x) := sf$

a2 : $senseBuff(x) := senseBuff(x) \cup \{sd\}$

end

Event *finish_sensing* $\widehat{=}$

//(shd) finishing sensing data process

extends *finish_sensing*

any

x

where

g1 : $sensingNDs = \{x\}$

g2 : $x \in dom(envSensedFlg)$

g3 : $envSensedFlg(x) = TRUE$

g4 : $ctlSensedFlg(x) = TRUE$

g5 : $ndBuff = \varnothing \wedge (sentDown \cup sentUp) = \varnothing$

then

a1 : $sensingNDs := \varnothing$

a2 : $ctlSensedFlg(x) := FALSE$

a3 : $envSensedFlg(x) := FALSE$

end

Event *create_dataPkt* $\widehat{=}$

//(ctl) creating a data packet

extends *create_dataPkt*

any

s

pkt
$fDes$
$data$
sno
nbh

where

g1 : $pkt \in PKT \wedge s \in ND \setminus Destination$
g2 : $pkt \notin ran(ndBuff \cup sentDown \cup sentUp)$
g3 : $s \notin sensingNDs \wedge pkt \notin (sensedPkts \cup floodedPkts)$
g4 : $s = initialSrcAddr(pkt)$
g5 : $s \in dom(floodTbl) \wedge pkt \notin floodTbl(s)$
g6 : $type(pkt) = DATA$
g7 : $fDes \in Destination$
g8 : $pkt \mapsto fDes \notin finalDestAddr$
g9 : $fDes \neq s$
g10 : $ctlSensedFlg(s) = TRUE$
g11 : $s \in dom(senseBuff) \wedge senseBuff \neq \varnothing \wedge data \in senseBuff(s)$
g12 : $pkt \notin dom(pktData)$
g13 : $sno = dataSeqNo(s) + 1 \wedge s \in dom(dataSeqNo)$
g14 : $nbh \in \mathbb{Z} \wedge nbh = -1$
g15 : $s \mapsto \{pkt \mapsto nbh\} \notin nbHops$
g16 : $pkt \notin dom(pktSeqNo)$
g17 : $pkt \notin dom(pktSrc)$
g18 : $pkt \notin dom(pktFwdr)$
g19 : $pkt \notin dom(pktNbHops)$
g20 : $pkt \notin dom(netDestAddr)$

then

a1 : $sensingNDs := sensingNDs \cup \{s\}$
a2 : $sensedPkts := sensedPkts \cup \{pkt\}$
a3 : $ndBuff := ndBuff \cup \{s \mapsto pkt\}$
a4 : $floodTbl(s) := floodTbl(s) \cup \{pkt\}$
a5 : $finalDestAddr := finalDestAddr \cup \{pkt \mapsto fDes\}$
a6 : $senseBuff(s) := senseBuff(s) \setminus \{data\}$
a7 : $pktData := pktData \cup \{pkt \mapsto data\}$
a8 : $dataSeqNo := dataSeqNo \lhd\!\!\!- \{s \mapsto sno\}$
a9 : $nbHops := nbHops \cup \{s \mapsto \{pkt \mapsto nbh\}\}$
a10 : $pktSeqNo := pktSeqNo \lhd\!\!\!- \{pkt \mapsto sno\}$
a11 : $pktSrc := pktSrc \cup \{pkt \mapsto s\}$
a12 : $pktFwdr := pktFwdr \cup \{pkt \mapsto s\}$
a13 : $pktNbHops := pktNbHops \cup \{pkt \mapsto nbh\}$
a14 : $netDestAddr := netDestAddr \cup \{pkt \mapsto BROADCAST\}$

end

Event *create_rreqPkt* $\widehat{=}$

//(ctl) creating a RREQ packet

extends *create_rreqPkt*

any

s

pkt

$fDes$

sno

nbh

where

g1 : $pkt \in PKT$

g2 : $pkt \notin ran(ndBuff \cup sentDown \cup sentUp)$

g3 : $pkt \notin (sensedPkts \cup floodedPkts)$

g4 : $s = initialSrcAddr(pkt)$

g5 : $s \in dom(floodTbl) \wedge pkt \notin floodTbl(s)$

g6 : $type(pkt) \in CONTROL$

g7 : $floodFlg(s) = TRUE$

g8 : $fDes \in ND$

g9 : $pkt \mapsto fDes \notin finalDestAddr$

g10 : $fDes \neq s$

g11 : $sno = floodSeqNo(s) + 1 \wedge s \in dom(floodSeqNo)$

g12 : $nbh \in \mathbb{Z} \wedge nbh = -1$

g13 : $s \mapsto \{pkt \mapsto nbh\} \notin nbHops$

g14 : $pkt \notin dom(pktSeqNo)$

g15 : $pkt \notin dom(pktSrc)$

g16 : $pkt \notin dom(pktFwdr)$

g17 : $pkt \notin dom(pktData)$

g18 : $pkt \notin dom(pktNbHops)$

g19 : $type(pkt) = RREQ$

g20 : $pkt \notin dom(netDestAddr)$

g21 : $rrepFlg(s) = FALSE$

then

a1 : $floodedPkts := floodedPkts \cup \{pkt\}$

a2 : $ndBuff := ndBuff \cup \{s \mapsto pkt\}$

a3 : $floodTbl(s) := floodTbl(s) \cup \{pkt\}$

a4 : $finalDestAddr := finalDestAddr \cup \{pkt \mapsto fDes\}$

a5 : $floodSeqNo := floodSeqNo \lhd\!\!\!- \{s \mapsto sno\}$

a6 : $nbHops := nbHops \cup \{s \mapsto \{pkt \mapsto nbh\}\}$

a7 : $pktSeqNo := pktSeqNo \lhd\!\!\!- \{pkt \mapsto sno\}$

a8 : $pktFwdr := pktFwdr \cup \{pkt \mapsto s\}$

a9 : $pktSrc := pktSrc \cup \{pkt \mapsto s\}$
a10 : $pktData := pktData \cup \{pkt \mapsto 0\}$
a11 : $pktNbHops := pktNbHops \cup \{pkt \mapsto nbh\}$
a12 : $netDestAddr := netDestAddr \cup \{pkt \mapsto BROADCAST\}$

end

Event *create_rrepPkt* $\widehat{=}$

//(ctl) creating a RREP packet

extends *create_rrepPkt*

any

s
pkt
$fDes$
sno
nbh
$rreq$

where

g1 : $pkt \in PKT$
g2 : $pkt \notin ran(ndBuff \cup sentDown \cup sentUp)$
g3 : $pkt \notin (sensedPkts \cup floodedPkts)$
g4 : $s = initialSrcAddr(pkt)$
g5 : $s \in dom(floodTbl) \wedge pkt \notin floodTbl(s)$
g6 : $type(pkt) \in CONTROL$
g7 : $floodFlg(s) = TRUE$
g8 : $fDes \in ND$
g9 : $pkt \mapsto fDes \notin finalDestAddr$
g10 : $fDes \neq s$
g11 : $sno = floodSeqNo(s) + 1 \wedge s \in dom(floodSeqNo)$
g12 : $nbh \in \mathbb{Z} \wedge nbh = -1$
g13 : $s \mapsto \{pkt \mapsto nbh\} \notin nbHops$
g14 : $pkt \notin dom(pktSeqNo)$
g15 : $pkt \notin dom(pktSrc)$
g16 : $pkt \notin dom(pktFwdr)$
g17 : $pkt \notin dom(pktData)$
g18 : $pkt \notin dom(pktNbHops)$
g19 : $type(pkt) = RREP$
g20 : $s \mapsto rreq \in rrepLists$
g21 : $pkt \notin dom(netDestAddr)$
g22 : $fDes = initialSrcAddr(rreq)$
g23 : $rrepFlg(s) = TRUE$

then

a1 : $floodedPkts := floodedPkts \cup \{pkt\}$

a2 : $ndBuff := ndBuff \cup \{s \mapsto pkt\}$
a3 : $floodTbl(s) := floodTbl(s) \cup \{pkt\}$
a4 : $finalDestAddr := finalDestAddr \cup \{pkt \mapsto fDes\}$
a5 : $floodSeqNo := floodSeqNo \lhd\!\!\!- \{s \mapsto sno\}$
a6 : $nbHops := nbHops \cup \{s \mapsto \{pkt \mapsto nbh\}\}$
a7 : $pktSeqNo := pktSeqNo \lhd\!\!\!- \{pkt \mapsto sno\}$
a8 : $pktFwdr := pktFwdr \cup \{pkt \mapsto s\}$
a9 : $pktSrc := pktSrc \cup \{pkt \mapsto s\}$
a10 : $pktData := pktData \cup \{pkt \mapsto 0\}$
a11 : $pktNbHops := pktNbHops \cup \{pkt \mapsto nbh\}$
a12 : $rrepLists := rrepLists \setminus \{s \mapsto rreq\}$
a13 : $netDestAddr := netDestAddr \cup \{pkt \mapsto BROADCAST\}$

end

Event *create_rrerPkt* $\widehat{=}$

//(ctl) create a RRER packet

extends *create_controlPkt*

any

s
pkt
$fDes$
sno
nbh
$rrer$
eND

where

g1 : $pkt \in PKT$
g2 : $pkt \notin ran(ndBuff \cup sentDown \cup sentUp)$
g3 : $pkt \notin (sensedPkts \cup floodedPkts)$
g4 : $s = initialSrcAddr(pkt)$
g5 : $s \in dom(floodTbl) \wedge pkt \notin floodTbl(s)$
g6 : $type(pkt) \in CONTROL$
g7 : $floodFlg(s) = TRUE$
g8 : $fDes \in ND$
g9 : $pkt \mapsto fDes \notin finalDestAddr$
g10 : $fDes \neq s$
g11 : $sno = floodSeqNo(s) + 1 \wedge s \in dom(floodSeqNo)$
g12 : $nbh \in \mathbb{Z} \wedge nbh = -1$
g13 : $s \mapsto \{pkt \mapsto nbh\} \notin nbHops$
g14 : $pkt \notin dom(pktSeqNo)$
g15 : $pkt \notin dom(pktSrc)$
g16 : $pkt \notin dom(pktFwdr)$

g17 : $pkt \notin dom(pktData)$
g18 : $pkt \notin dom(pktNbHops)$
g19 : $type(pkt) \neq RREP \wedge type(pkt) \neq RREQ$
g20 : $pkt \notin dom(netDestAddr)$
g21 : $rrepFlg(s) = FALSE$
g22 : $eND \in ND$
g23 : $s \mapsto rrer \in rrerLists$
g24 : $rrer \in dom(initialSrcAddr) \wedge fDes = initialSrcAddr(rrer)$
g25 : $s \mapsto eND \in fwdRouteTbl$
g26 : $pkt \notin dom(pktErrND)$
g27 : $rrerFlg(s) = TRUE$

then

a1 : $floodedPkts := floodedPkts \cup \{pkt\}$
a2 : $ndBuff := ndBuff \cup \{s \mapsto pkt\}$
a3 : $floodTbl(s) := floodTbl(s) \cup \{pkt\}$
a4 : $finalDestAddr := finalDestAddr \cup \{pkt \mapsto fDes\}$
a5 : $floodSeqNo := floodSeqNo \lhd\!\!\!- \{s \mapsto sno\}$
a6 : $nbHops := nbHops \cup \{s \mapsto \{pkt \mapsto nbh\}\}$
a7 : $pktSeqNo := pktSeqNo \lhd\!\!\!- \{pkt \mapsto sno\}$
a8 : $pktFwdr := pktFwdr \cup \{pkt \mapsto s\}$
a9 : $pktSrc := pktSrc \cup \{pkt \mapsto s\}$
a10 : $pktData := pktData \cup \{pkt \mapsto 0\}$
a11 : $pktNbHops := pktNbHops \cup \{pkt \mapsto nbh\}$
a12 : $netDestAddr := netDestAddr \cup \{pkt \mapsto BROADCAST\}$
a13 : $rrerLists := rrerLists \setminus \{s \mapsto rrer\}$
a14 : $pktErrND := pktErrND \cup \{pkt \mapsto eND\}$

end

Event *start_tx_dataPkt_bct* $\widehat{=}$

//(ctl) starting broadcasting a data packet to a channel

extends *start_tx_dataPkt_bct*

any

x
pkt
nbh

where

g1 : $x \mapsto pkt \in ndBuff$
g2 : $x \mapsto pkt \notin sentDown$
g3 : $x \mapsto pkt \notin sentUp$
g4 : $x \mapsto \{pkt \mapsto nbh\} \in nbHops$
g5 : $pkt \in dom(pktFwdr)$
g6 : $pkt \in dom(pktNbHops)$

g7 : $type(pkt) = DATA$
g8 : $ran(\{x\} \lhd dom(fwdNextND)) = \varnothing$

then

a1 : $ndBuff := ndBuff \setminus \{x \mapsto pkt\}$
a2 : $sentDown := sentDown \cup \{x \mapsto pkt\}$
a3 : $pktFwdr := pktFwdr \lhd\!\!\!- \{pkt \mapsto x\}$
a4 : $pktNbHops := pktNbHops \lhd\!\!\!- \{pkt \mapsto nbh\}$
a5 : $nbHops := nbHops \setminus \{x \mapsto \{pkt \mapsto nbh\}\}$

end

Event *start_tx_dataPkt_fwd* $\widehat{=}$

//(ctl) starting unicasting a data packet to a channel via route

extends *start_tx_dataPkt_fwd*

any

x
pkt
nbh
s
nxt

where

g1 : $x \mapsto pkt \in ndBuff$
g2 : $x \mapsto pkt \notin sentDown$
g3 : $x \mapsto pkt \notin sentUp$
g4 : $x \mapsto \{pkt \mapsto nbh\} \in nbHops$
g5 : $pkt \in dom(pktFwdr)$
g6 : $pkt \in dom(pktNbHops)$
g7 : $type(pkt) = DATA$
g8 : $ran(\{x\} \lhd dom(fwdNextND)) \neq \varnothing$
$\wedge\ s \in ran(\{x\} \lhd dom(fwdNextND))$
g9 : $nxt = fwdNextND(x \mapsto s)$

then

a1 : $ndBuff := ndBuff \setminus \{x \mapsto pkt\}$
a2 : $sentDown := sentDown \cup \{x \mapsto pkt\}$
a3 : $pktFwdr := pktFwdr \lhd\!\!\!- \{pkt \mapsto x\}$
a4 : $pktNbHops := pktNbHops \lhd\!\!\!- \{pkt \mapsto nbh\}$
a5 : $nbHops := nbHops \setminus \{x \mapsto \{pkt \mapsto nbh\}\}$
a13 : $netDestAddr := netDestAddr \lhd\!\!\!- \{pkt \mapsto nxt\}$

end

Event *start_tx_rrep* $\widehat{=}$

//(ctl) starting transmiting a RREQ packet to a channel

extends *start_tx_rrep*

any

x

pkt

nbh

$lsno$

s

nxt

where

g1 : $x \mapsto pkt \in ndBuff$

g2 : $x \mapsto pkt \notin sentDown$

g3 : $x \mapsto pkt \notin sentUp$

g4 : $x \mapsto \{pkt \mapsto nbh\} \in nbHops$

g5 : $pkt \in dom(pktFwdr)$

g6 : $pkt \in dom(pktNbHops)$

g7 : $type(pkt) \neq RREQ$

g8 : $lsno \in \mathbb{N}$

g9 : $type(pkt) = RREP$

g10 : $x \in dom(rrepSeqNo) \wedge lsno = rrepSeqNo(x) + 1$

g11 : $x \mapsto s \in dom(bwdNextND) \wedge nxt = bwdNextND(x \mapsto s)$

g12 : $pkt \in dom(netDestAddr)$

g13 : $pkt \in dom(finalDestAddr) \wedge s = finalDestAddr(pkt)$

then

a1 : $ndBuff := ndBuff \setminus \{x \mapsto pkt\}$

a2 : $sentDown := sentDown \cup \{x \mapsto pkt\}$

a3 : $pktFwdr := pktFwdr \lhd\!\!\!- \{pkt \mapsto x\}$

a4 : $pktNbHops := pktNbHops \lhd\!\!\!- \{pkt \mapsto nbh\}$

a5 : $nbHops := nbHops \setminus \{x \mapsto \{pkt \mapsto nbh\}\}$

a6 : $netSeqNo(pkt) := lsno$

a7 : $rrepSeqNo := rrepSeqNo \lhd\!\!\!- \{x \mapsto lsno\}$

a8 : $netDestAddr := netDestAddr \lhd\!\!\!- \{pkt \mapsto nxt\}$

end

Event *start_tx_rreq* $\widehat{=}$

//(ctl) starting transmiting a RREP packet to a channel

extends *start_tx_rreq*

any

x

pkt

nbh

$lsno$

where

g1 : $x \mapsto pkt \in ndBuff$

g2 : $x \mapsto pkt \notin sentDown$

g3 : $x \mapsto pkt \notin sentUp$
g4 : $x \mapsto \{pkt \mapsto nbh\} \in nbHops$
g5 : $pkt \in dom(pktFwdr)$
g6 : $pkt \in dom(pktNbHops)$
g7 : $type(pkt) = RREQ$
g8 : $x \in dom(linkSeqNo) \wedge lsno = linkSeqNo(x) + 1$

then

a1 : $ndBuff := ndBuff \setminus \{x \mapsto pkt\}$
a2 : $sentDown := sentDown \cup \{x \mapsto pkt\}$
a3 : $pktFwdr := pktFwdr \lhd\!\!\!- \{pkt \mapsto x\}$
a4 : $pktNbHops := pktNbHops \lhd\!\!\!- \{pkt \mapsto nbh\}$
a5 : $nbHops := nbHops \setminus \{x \mapsto \{pkt \mapsto nbh\}\}$
a6 : $linkSeqNo := linkSeqNo \lhd\!\!\!- \{x \mapsto lsno\}$
a7 : $netSeqNo(pkt) := lsno$

end

Event *start_tx_rrer* $\widehat{=}$

//(ctl) starting transmiting a RRER packet to a channel

extends *start_tx_controlPkt*

any

x
pkt
nbh
$lsno$
nxt
$fDes$

where

g1 : $x \mapsto pkt \in ndBuff$
g2 : $x \mapsto pkt \notin sentDown$
g3 : $x \mapsto pkt \notin sentUp$
g4 : $x \mapsto \{pkt \mapsto nbh\} \in nbHops$
g5 : $pkt \in dom(pktFwdr)$
g6 : $pkt \in dom(pktNbHops)$
g7 : $type(pkt) \neq RREQ$
g8 : $lsno \in \mathbb{N}$
g9 : $type(pkt) \neq RREP \wedge type(pkt) \neq RREQ$
g10 : $nxt \in ND$
g11 : $pkt \in dom(finalDestAddr) \wedge fDes = finalDestAddr(pkt)$
g12 : $ran(\{x\} \lhd dom(bwdNextND)) \neq \varnothing$
$\wedge\ fDes \in ran(\{x\} \lhd dom(bwdNextND))$
g13 : $nxt = bwdNextND(x \mapsto fDes)$
g14 : $type(pkt) = RRER$

then

a1 : $ndBuff := ndBuff \setminus \{x \mapsto pkt\}$
a2 : $sentDown := sentDown \cup \{x \mapsto pkt\}$
a3 : $pktFwdr := pktFwdr \lhd\!\!\!- \{pkt \mapsto x\}$
a4 : $pktNbHops := pktNbHops \lhd\!\!\!- \{pkt \mapsto nbh\}$
a5 : $nbHops := nbHops \setminus \{x \mapsto \{pkt \mapsto nbh\}\}$
a6 : $netSeqNo(pkt) := lsno$
a7 : $netDestAddr := netDestAddr \lhd\!\!\!- \{pkt \mapsto nxt\}$

end

Event *send_down* $\widehat{=}$

//(shd) sending a packet down to a channel

extends *send_down*

any

cn
pkt
sno
src
$data$
$fwdr$
nbh
nxt

where

g1 : $cn \mapsto pkt \in sentDown$
g2 : $cn \mapsto pkt \notin channel$
g3 : $pkt \in dom(pktSeqNo) \wedge sno = pktSeqNo(pkt)$
g4 : $pkt \in dom(pktSrc) \wedge src = pktSrc(pkt)$
g5 : $pkt \in dom(pktFwdr) \wedge fwdr = pktFwdr(pkt)$
g6 : $pkt \in dom(pktData) \wedge data = pktData(pkt)$
g7 : $pkt \in dom(pktNbHops) \wedge nbh = pktNbHops(pkt)$
g8 : $pkt \notin dom(vPktSeqNo)$
g9 : $pkt \notin dom(vPktFwdr)$
g10 : $pkt \notin dom(vPktSrc)$
g11 : $pkt \notin dom(vPktData)$
g12 : $pkt \notin dom(vPktNbHops)$
g13 : $pkt \in dom(netDestAddr) \wedge nxt = netDestAddr(pkt)$
g14 : $pkt \notin dom(envDestAddr) \wedge nxt \in ND \cup \{BROADCAST\}$

then

a1 : $channel := channel \cup \{cn \mapsto pkt\}$
a2 : $vPktSeqNo := vPktSeqNo \cup \{pkt \mapsto sno\}$
a3 : $vPktSrc := vPktSrc \cup \{pkt \mapsto src\}$
a4 : $vPktFwdr := vPktFwdr \cup \{pkt \mapsto fwdr\}$

a5 : $vPktData := vPktData \cup \{pkt \mapsto data\}$
a6 : $vPktNbHops := vPktNbHops \cup \{pkt \mapsto nbh\}$
a7 : $pktSeqNo := \{pkt\} \lhd pktSeqNo$
a8 : $pktSrc := \{pkt\} \lhd pktSrc$
a9 : $pktFwdr := \{pkt\} \lhd pktFwdr$
a10 : $pktData := \{pkt\} \lhd pktData$
a11 : $pktNbHops := \{pkt\} \lhd pktNbHops$
a12 : $envDestAddr := envDestAddr \cup \{pkt \mapsto nxt\}$

end

Event *find_neighbours* $\widehat{=}$

//(env) channel finds the neighbour list of the forwarder

extends *find_neighbours*

any

s
nbs
f
pkt
nxt

where

g1 : $channel \neq \varnothing$
g2 : $pkt \in ran(channel)$
g3 : $s = initialSrcAddr(pkt)$
g4 : $f \mapsto pkt \in channel$
g5 : $nbs \subseteq ND \wedge nbs = wsnLinks[\{f\}] \wedge nbs \neq \varnothing$
g6 : $envNeighbours = \varnothing$
g7 : $pkt \in dom(netDestAddr) \wedge nxt = netDestAddr(pkt)$
g8 : $nxt = BROADCAST$

then

a1 : $envNeighbours := envNeighbours \cup (\{pkt\} \times nbs)$

end

Event *assign_forwarder* $\widehat{=}$

//(env) channel assigns specific forwarder

extends *assign_forwarder*

any

s
nb
pkt
f
nbs
nxt

where

g1 : $channel \neq \varnothing$
g2 : $pkt \in ran(channel)$
g3 : $s = initialSrcAddr(pkt)$
g4 : $f \mapsto pkt \in channel$
g5 : $nbs = wsnLinks[\{f\}] \wedge nbs \neq \varnothing$
g6 : $nb \in nbs$
g7 : $pkt \in dom(netDestAddr) \wedge nxt = netDestAddr(pkt)$
g8 : $nxt \neq BROADCAST$
g9 : $nb = nxt$

then

a1 : $envNeighbours := envNeighbours \cup \{pkt \mapsto nb\}$

end

Event *send_up* $\widehat{=}$

//(env) channel sends the neighbour information

extends *send_up*

any

pkt
$nbrs$
f
sno
src
$fwdr$
$data$
nbh
nxt

where

g1 : $pkt \in ran(sentDown)$
g2 : $f \mapsto pkt \in sentDown$
g3 : $f \mapsto pkt \notin sentUp$
g4 : $f \mapsto pkt \in channel$
g5 : $nbrs \in \{n | n \in (\mathbb{P}(ND)) \cup \{\{FAILED\_XMIT\}\}\}$
g6 : $nbrs \neq \varnothing$
g7 : $ran(\{pkt\} \lhd envNeighbours) \in (\mathbb{P}(ND) \cup \{\{FAILED\_XMIT\}\})$
g8 : $nbrs = ran(\{pkt\} \lhd envNeighbours) \wedge nbrs \neq \varnothing$
g9 : $pkt \notin dom(ctlNeighbours)$
g10 : $pkt \in dom(vPktSeqNo) \wedge sno = vPktSeqNo(pkt)$
g11 : $pkt \in dom(vPktSrc) \wedge src = vPktSrc(pkt)$
g12 : $pkt \in dom(vPktFwdr) \wedge fwdr = vPktFwdr(pkt)$
g13 : $pkt \in dom(vPktData) \wedge data = vPktData(pkt)$
g14 : $pkt \in dom(vPktNbHops) \wedge nbh = vPktNbHops(pkt)$
g15 : $pkt \notin dom(pktSeqNo)$

g16 : $pkt \notin dom(pktSrc)$
g17 : $pkt \notin dom(pktFwdr)$
g18 : $pkt \notin dom(pktData)$
g19 : $pkt \notin dom(pktNbHops)$
g20 : $pkt \mapsto nxt \in envDestAddr$

then

a1 : $ctlNeighbours := ctlNeighbours \cup (\{pkt\} \times nbrs)$
a2 : $envNeighbours := \varnothing$
a3 : $channel := channel \setminus \{f \mapsto pkt\}$
a4 : $sentDown := sentDown \setminus \{f \mapsto pkt\}$
a5 : $sentUp := sentUp \cup \{f \mapsto pkt\}$
a6 : $pktSeqNo := pktSeqNo \cup \{pkt \mapsto sno\}$
a7 : $pktSrc := pktSrc \cup \{pkt \mapsto src\}$
a8 : $pktFwdr := pktFwdr \cup \{pkt \mapsto fwdr\}$
a9 : $pktData := pktData \cup \{pkt \mapsto data\}$
a10 : $pktNbHops := pktNbHops \cup \{pkt \mapsto nbh\}$
a11 : $vPktSeqNo := \{pkt\} \lhd vPktSeqNo$
a12 : $vPktSrc := \{pkt\} \lhd vPktSrc$
a13 : $vPktFwdr := \{pkt\} \lhd vPktFwdr$
a14 : $vPktData := \{pkt\} \lhd vPktData$
a15 : $vPktNbHops := \{pkt\} \lhd vPktNbHops$
a16 : $envDestAddr := envDestAddr \setminus \{pkt \mapsto nxt\}$

end

Event *receive_dataPkt* $\widehat{=}$

//(ctl) each neighbour node receives a data packet from channel

extends *receive_dataPkt*

any

s
nb
pkt
nbh
f

where

g1 : $pkt \in ran(sentUp \cup sentDown)$
g2 : $pkt \mapsto nb \in ctlNeighbours$
g3 : $nb \mapsto pkt \notin ndBuff$
g4 : $nb \notin dom(sentUp \cup sentDown)$
g5 : $s = initialSrcAddr(pkt) \wedge s \neq nb$
g6 : $pkt \in dom(finalDestAddr) \wedge nb \neq finalDestAddr(pkt)$
g7 : $nb \in dom(floodTbl) \wedge pkt \notin floodTbl(nb)$
g8 : $pkt \in dom(pktNbHops) \wedge nbh = pktNbHops(pkt) + 1$

g9 : $pkt \mapsto f \in pktFwdr$
g10 : $type(pkt) = DATA$

then

a1 : $ndBuff := ndBuff \cup \{nb \mapsto pkt\}$
a2 : $ctlNeighbours := ctlNeighbours \setminus \{pkt \mapsto nb\}$
a3 : $floodTbl(nb) := floodTbl(nb) \cup \{pkt\}$
a4 : $nbHops := nbHops \cup \{nb \mapsto \{pkt \mapsto nbh\}\}$

end

Event *receive_dup_dataPkt* $\widehat{=}$

//(ctl) each neighbour node including sink receives a duplicated data packet from channel

extends *receive_dup_dataPkt*

any

nb
pkt
f

where

g1 : $pkt \in dom(ctlNeighbours)$
g2 : $pkt \mapsto nb \in ctlNeighbours$
g3 : $nb \in dom(floodTbl) \wedge pkt \in floodTbl(nb)$
g4 : $pkt \mapsto f \in pktFwdr$
g5 : $type(pkt) = DATA$

then

a1 : $ctlNeighbours := ctlNeighbours \setminus \{pkt \mapsto nb\}$

end

Event *receive_rreqPkt* $\widehat{=}$

//(ctl) each neighbour node receives a RREQ packet from channel

extends *receive_rreqPkt*

any

s
nb
pkt
nbh
f

where

g1 : $pkt \in ran(sentUp \cup sentDown)$
g2 : $pkt \mapsto nb \in ctlNeighbours$
g3 : $nb \mapsto pkt \notin ndBuff$
g4 : $nb \notin dom(sentUp \cup sentDown)$
g5 : $s = initialSrcAddr(pkt) \wedge s \neq nb$

g6 : $pkt \in dom(finalDestAddr) \wedge nb \neq finalDestAddr(pkt)$
g7 : $nb \in dom(floodTbl) \wedge pkt \notin floodTbl(nb)$
g8 : $pkt \in dom(pktNbHops) \wedge nbh = pktNbHops(pkt) + 1$
g9 : $pkt \mapsto f \in pktFwdr$
g10 : $type(pkt) \in CONTROL$
g11 : $f \in ND \wedge nb \in ND$
g12 : $f \mapsto nb \notin updateNbrs$
g13 : $type(pkt) = RREQ$

then

a1 : $ndBuff := ndBuff \cup \{nb \mapsto pkt\}$
a2 : $ctlNeighbours := ctlNeighbours \setminus \{pkt \mapsto nb\}$
a3 : $floodTbl(nb) := floodTbl(nb) \cup \{pkt\}$
a4 : $nbHops := nbHops \cup \{nb \mapsto \{pkt \mapsto nbh\}\}$
a5 : $updateNbrs := updateNbrs \cup \{f \mapsto nb\}$
description

end

Event *receive_dup_rreqPkt* $\widehat{=}$

//(ctl) each neighbour node including sink receives a duplicated RREQ packet from channel

extends *receive_dup_rreqPkt*

any

nb
pkt
f

where

g1 : $pkt \in dom(ctlNeighbours)$
g2 : $pkt \mapsto nb \in ctlNeighbours$

g3 : $nb \in dom(floodTbl) \wedge pkt \in floodTbl(nb)$
g4 : $pkt \mapsto f \in pktFwdr$
g5 : $type(pkt) \neq DATA$
g6 : $f \in ND \wedge nb \in ND$
g7 : $f \mapsto nb \notin updateNbrs$
g8 : $type(pkt) = RREQ$

then

a1 : $ctlNeighbours := ctlNeighbours \setminus \{pkt \mapsto nb\}$

end

Event *receive_rrepPkt* $\widehat{=}$

//(ctl) each neighbour node receives a RREP packet from channel

extends *receive_rrepPkt*

any

s

nb

pkt

nbh

f

where

g1 : $pkt \in ran(sentUp \cup sentDown)$

g2 : $pkt \mapsto nb \in ctlNeighbours$

g3 : $nb \mapsto pkt \notin ndBuff$

g4 : $nb \notin dom(sentUp \cup sentDown)$

g5 : $s = initialSrcAddr(pkt) \wedge s \neq nb$

g6 : $pkt \in dom(finalDestAddr) \wedge nb \neq finalDestAddr(pkt)$

g7 : $nb \in dom(floodTbl) \wedge pkt \notin floodTbl(nb)$

g8 : $pkt \in dom(pktNbHops) \wedge nbh = pktNbHops(pkt) + 1$

g9 : $pkt \mapsto f \in pktFwdr$

g10 : $type(pkt) \in CONTROL$

g11 : $f \in ND \wedge nb \in ND$

g12 : $f \mapsto nb \notin updateNbrs$

g13 type(pkt) = RREP

then

a1 : $ndBuff := ndBuff \cup \{nb \mapsto pkt\}$

a2 : $ctlNeighbours := ctlNeighbours \setminus \{pkt \mapsto nb\}$

a3 : $floodTbl(nb) := floodTbl(nb) \cup \{pkt\}$

a4 : $nbHops := nbHops \cup \{nb \mapsto \{pkt \mapsto nbh\}\}$

a5 : $updateNbrs := updateNbrs \cup \{f \mapsto nb\}$

description

end

Event *receive_dup_rrepPkt* $\widehat{=}$

//(ctl) each neighbour node including sink receives a duplicated RREP packet from channel

extends *receive_dup_rrepPkt*

any

nb

pkt

f

where

g1 : $pkt \in dom(ctlNeighbours)$

g2 : $pkt \mapsto nb \in ctlNeighbours$

g3 : $nb \in dom(floodTbl) \wedge pkt \in floodTbl(nb)$

g4 : $pkt \mapsto f \in pktFwdr$

g5 : $type(pkt) \neq DATA$

g6 : $f \in ND \wedge nb \in ND$
g7 : $f \mapsto nb \notin updateNbrs$
g8 : $type(pkt) = RREP$

then

a1 : $ctlNeighbours := ctlNeighbours \setminus \{pkt \mapsto nb\}$

end

Event *receive_rrerPkt* $\widehat{=}$

//(ctl) each neighbour node receives a RRER pacekt from channel

extends *receive_controlPkt*

any

s
nb
pkt
nbh
f

where

g1 : $pkt \in ran(sentUp \cup sentDown)$
g2 : $pkt \mapsto nb \in ctlNeighbours$
g3 : $nb \mapsto pkt \notin ndBuff$
g4 : $nb \notin dom(sentUp \cup sentDown)$
g5 : $s = initialSrcAddr(pkt) \wedge s \neq nb$
g6 : $pkt \in dom(finalDestAddr) \wedge nb \neq finalDestAddr(pkt)$
g7 : $nb \in dom(floodTbl) \wedge pkt \notin floodTbl(nb)$
g8 : $pkt \in dom(pktNbHops) \wedge nbh = pktNbHops(pkt) + 1$
g9 : $pkt \mapsto f \in pktFwdr$
g10 : $type(pkt) \in CONTROL$
g11 : $f \in ND \wedge nb \in ND$
g12 : $f \mapsto nb \notin updateNbrs$
g13 : $type(pkt) \neq RREQ \wedge type(pkt) \neq RREP$
g14 : $type(pkt) = RRER$

then

a1 : $ndBuff := ndBuff \cup \{nb \mapsto pkt\}$
a2 : $ctlNeighbours := ctlNeighbours \setminus \{pkt \mapsto nb\}$
a3 : $floodTbl(nb) := floodTbl(nb) \cup \{pkt\}$
a4 : $nbHops := nbHops \cup \{nb \mapsto \{pkt \mapsto nbh\}\}$
a5 : $updateNbrs := updateNbrs \cup \{f \mapsto nb\}$
description

end

Event *receive_dup_rrerPkt* $\widehat{=}$

//(ctl) each neighbour node including sink receives a duplicated RRER packet from channel

extends *receive_dup_controlPkt*

any

nb

pkt

f

where

g1 : $pkt \in dom(ctlNeighbours)$

g2 : $pkt \mapsto nb \in ctlNeighbours$

g3 : $nb \in dom(floodTbl) \wedge pkt \in floodTbl(nb)$

g4 : $pkt \mapsto f \in pktFwdr$

g5 : $type(pkt) \neq DATA$

g6 : $f \in ND \wedge nb \in ND$

g7 : $f \mapsto nb \notin updateNbrs$

g8 : $type(pkt) \neq RREQ \wedge type(pkt) \neq RREP$

g9 : $type(pkt) = RRER$

then

a1 : $ctlNeighbours := ctlNeighbours \setminus \{pkt \mapsto nb\}$

end

Event *dest_recv_dataPkt* $\widehat{=}$

//(ct) destination receives a data packet

extends *dest_recv_dataPkt*

any

s

des

pkt

$data$

f

where

g1 : $pkt \in PKT$

g2 : $pkt \in dom(finalDestAddr) \wedge des = finalDestAddr(pkt)$

g3 : $des \in dom(destBuff)$

g4 : $pkt \in ran(sentUp \cup sentDown) \setminus destBuff(des)$

g5 : $s = initialSrcAddr(pkt)$

g6 : $\{pkt\} \lhd ctlNeighbours \neq \varnothing$

g7 : $ran(\{pkt\} \lhd ctlNeighbours) \neq \{FAILED\_XMIT\}$

g8 : $des \in (ran(\{pkt\} \lhd ctlNeighbours))$

g9 : $des \in dom(recvedData)$

g10 : $pkt \in dom(pktData) \wedge data = pktData(pkt)$

g11 : $type(pkt) = DATA$

g12 : $pkt \mapsto f \in pktFwdr$

then

a1 : $destBuff(des) := destBuff(des) \cup \{pkt\}$
a2 : $ctlNeighbours := ctlNeighbours \setminus \{pkt \mapsto des\}$
a3 : $floodTbl(des) := floodTbl(des) \cup \{pkt\}$
a4 : $recvedData(des) := recvedData(des) \cup \{data\}$

end

Event *dest_recv_rreqPkt* $\widehat{=}$

//(ctl) destination receives a RREQ packet

extends *dest_recv_rreqPkt*

any

s
des
pkt
f

where

g1 : $pkt \in PKT$
g2 : $pkt \in dom(finalDestAddr) \wedge des = finalDestAddr(pkt)$
g3 : $des \in dom(destBuff)$
g4 : $pkt \in ran(sentUp \cup sentDown) \setminus destBuff(des)$
g5 : $s = initialSrcAddr(pkt)$
g6 : $\{pkt\} \lhd ctlNeighbours \neq \varnothing$
g7 : $ran(\{pkt\} \lhd ctlNeighbours) \neq \{FAILED\_XMIT\}$
g8 : $des \in (ran(\{pkt\} \lhd ctlNeighbours))$
g9 : $type(pkt) \in CONTROL$
g10 : $pkt \mapsto f \in pktFwdr$
g11 : $type(pkt) \neq DATA$
g12 : $f \in ND \wedge des \in ND$
g13 : $f \mapsto des \notin updateNbrs$
g14 : $type(pkt) = RREQ$
g15 : $des \mapsto pkt \notin rrepLists$

then

a1 : $destBuff(des) := destBuff(des) \cup \{pkt\}$
a2 : $ctlNeighbours := ctlNeighbours \setminus \{pkt \mapsto des\}$
a3 : $floodTbl(des) := floodTbl(des) \cup \{pkt\}$
a4 : $updateNbrs := updateNbrs \cup \{f \mapsto des\}$
a5 : $rrepLists := rrepLists \cup \{des \mapsto pkt\}$

end

Event *dest_recv_rrepPkt* $\widehat{=}$

//(ctl) destination receives a RREP packet

extends *dest_recv_rrepPkt*

any

s

des

pkt

f

where

g1 : $pkt \in PKT$

g2 : $pkt \in dom(finalDestAddr) \wedge des = finalDestAddr(pkt)$

g3 : $des \in dom(destBuff)$

g4 : $pkt \in ran(sentUp \cup sentDown) \setminus destBuff(des)$

g5 : $s = initialSrcAddr(pkt)$

g6 : $\{pkt\} \lhd ctlNeighbours \neq \varnothing$

g7 : $ran(\{pkt\} \lhd ctlNeighbours) \neq \{FAILED\_XMIT\}$

g8 : $des \in (ran(\{pkt\} \lhd ctlNeighbours))$

g9 : $type(pkt) \in CONTROL$

g10 : $pkt \mapsto f \in pktFwdr$

g11 : $type(pkt) \neq DATA$

g12 : $f \in ND \wedge des \in ND$

g13 : $f \mapsto des \notin updateNbrs$

g14 : $type(pkt) = RREP$

then

a1 : $destBuff(des) := destBuff(des) \cup \{pkt\}$

a2 : $ctlNeighbours := ctlNeighbours \setminus \{pkt \mapsto des\}$

a3 : $floodTbl(des) := floodTbl(des) \cup \{pkt\}$

a4 : $updateNbrs := updateNbrs \cup \{f \mapsto des\}$

end

Event *dest_recv_rrerpkt* $\widehat{=}$

//(ctl) destination receives a RRER packet

extends *dest_recv_controlPkt*

any

s

des

pkt

f

where

g1 : $pkt \in PKT$

g2 : $pkt \in dom(finalDestAddr) \wedge des = finalDestAddr(pkt)$

g3 : $des \in dom(destBuff)$

g4 : $pkt \in ran(sentUp \cup sentDown) \setminus destBuff(des)$

g5 : $s = initialSrcAddr(pkt)$

g6 : $\{pkt\} \lhd ctlNeighbours \neq \varnothing$

g7 : $ran(\{pkt\} \lhd ctlNeighbours) \neq \{FAILED\_XMIT\}$

g8 : $des \in (ran(\{pkt\} \lhd ctlNeighbours))$

g9 : $type(pkt) \in CONTROL$

g10 : $pkt \mapsto f \in pktFwdr$

g11 : $type(pkt) \neq DATA$

g12 : $f \in ND \wedge des \in ND$

g13 : $f \mapsto des \notin updateNbrs$

g14 : $type(pkt) \neq RREQ \wedge type(pkt) \neq RREP$

then

a1 : $destBuff(des) := destBuff(des) \cup \{pkt\}$

a2 : $ctlNeighbours := ctlNeighbours \setminus \{pkt \mapsto des\}$

a3 : $floodTbl(des) := floodTbl(des) \cup \{pkt\}$

a4 : $updateNbrs := updateNbrs \cup \{f \mapsto des\}$

end

Event *actuating* $\widehat{=}$

//(ctl) alerting emergency: sensed data is above safety threshold

extends *actuating*

any

act

data

where

g1 : $act \in Actuators$

g2 : $act \in dom(recvedData)$

g3 : $data \in recvedData(act)$

g4 : $data \geq safetyThreshold$

g5 : $emergencyAlert(act) = FALSE$

then

a1 : $recvedData(act) := recvedData(act) \setminus \{data\}$

a2 : $emergencyAlert(act) := TRUE$

end

Event *no_actuating* $\widehat{=}$

//(ctl) not alerting emergency sensed data is below safety threshold

extends *no_actuating*

any

act

data

where

g1 : $act \in Actuators$

g2 : $act \in dom(recvedData)$

g3 : $data \in recvedData(act)$

g4 : $data < safetyThreshold$

then

a1 : $recvedData(act) := recvedData(act) \setminus \{data\}$

end

Event *reset_actuatingStatus* $\widehat{=}$

//(ctl) resetting emergency alert status

extends *reset_actuatingStatus*

any

act

where

g1 : $act \in dom(emergencyAlert)$

g2 : $emergencyAlert(act) = TRUE$

then

a1 : $emergencyAlert(act) := FALSE$

end

Event *add_bwdRouteEntry* $\widehat{=}$

//(ctl) add new entry into the backwards route table

extends *add_bwdRouteEntry*

any

x

y

s

pkt

sNo

$hCnt$

where

g1 : $updateNbrs \neq \varnothing$

//there is some nodes that have to update thier neighbour table

g2 : $x \in ND \wedge y \in ND \wedge x \mapsto y \in updateNbrs$

g3 : $pkt \in dom(initialSrcAddr) \wedge s = initialSrcAddr(pkt)$

g4 : $y \mapsto s \notin bwdRouteTbl$

//in case these new pairs don't appear in the neighbour table

g5 : $pkt \mapsto x \in pktFwdr$

g6 : $sNo = netSeqNo(pkt) \wedge y \mapsto x \mapsto sNo \notin bwdSeqNo$

g7 : $hCnt \in \mathbb{N} \wedge pkt \in dom(pktNbHops)$

$\wedge\ hCnt = pktNbHops(pkt) + 1 \wedge y \mapsto x \mapsto hCnt \notin bwdHopCnt$

g8 : $ctlNeighbours = \varnothing$

g9 : $type(pkt) = RREQ$

g10 : $type(pkt) = RREQ$

g12 : $pkt \in ran(sentUp)$

then

a1 : $bwdRouteTbl := bwdRouteTbl \cup \{y \mapsto s\}$

a2 : $bwdNextND := bwdNextND \cup \{y \mapsto s \mapsto x\}$

a3 : $bwdSeqNo := bwdSeqNo \cup \{y \mapsto s \mapsto sNo\}$

a4 : $bwdHopCnt := bwdHopCnt \cup \{y \mapsto s \mapsto hCnt\}$

a5 : $updateNbrs := updateNbrs \setminus \{x \mapsto y\}$

end

Event *add_fwdrouteEntry* $\widehat{=}$

//(ctl) add new entry into the fordwards route table

extends *add_fwdrouteEntry*

any

x

y

s

pkt

sNo

$hCnt$

where

g1 : $updateNbrs \neq \varnothing$

//there is some nodes that have to update thier neighbour table

g2 : $x \in ND \wedge y \in ND \wedge x \mapsto y \in updateNbrs$

g3 : $pkt \in dom(initialSrcAddr) \wedge s = initialSrcAddr(pkt)$

// in case these new pairs don't appear in the neighbour table

g4 : $pkt \mapsto x \in pktFwdr$

g5 : $sNo = netSeqNo(pkt) \wedge y \mapsto x \mapsto sNo \notin bwdSeqNo$

g6 : $hCnt \in \mathbb{N} \wedge pkt \in dom(pktNbHops) \wedge hCnt = pktNbHops(pkt) + 1$
$\wedge\, y \mapsto x \mapsto hCnt \notin bwdHopCnt$

g7 : $ctlNeighbours = \varnothing$

g8 : $type(pkt) \neq DATA \wedge type(pkt) \neq RREQ$

g9 : $y \mapsto s \notin fwdRouteTbl$

g10 : $type(pkt) = RREP$

g11 : $pkt \in ran(sentUp)$

then

a1 : $updateNbrs := updateNbrs \setminus \{x \mapsto y\}$

a2 : $fwdRouteTbl := fwdRouteTbl \cup \{y \mapsto s\}$

a3 : $fwdNextND := fwdNextND \cup \{y \mapsto s \mapsto x\}$

a4 : $fwdSeqNo := fwdSeqNo \cup \{y \mapsto s \mapsto sNo\}$

a5 : $fwdHopCnt := fwdHopCnt \cup \{y \mapsto s \mapsto hCnt\}$

end

Event *invalidate_neighbour* $\widehat{=}$

//(ctl) invalidating a dead neighbour

extends *add_routeEntry3*

any

x

y

s

pkt

sNo

$hCnt$

eND

where

g1 : $updateNbrs \neq \varnothing$

//there is some nodes that have to update thier neighbour table

g2 : $x \in ND \wedge y \in ND \wedge x \mapsto y \in updateNbrs$

g3 : $pkt \in dom(initialSrcAddr) \wedge s = initialSrcAddr(pkt)$

// in case these new pairs don't appear in the neighbour table

g5 : $pkt \mapsto x \in pktFwdr$

g6 : $sNo = netSeqNo(pkt) \wedge y \mapsto x \mapsto sNo \notin bwdSeqNo$

g7 : $hCnt \in \mathbb{N} \wedge pkt \in dom(pktNbHops)$

$\wedge\ hCnt = pktNbHops(pkt) + 1 \wedge y \mapsto x \mapsto hCnt \notin bwdHopCnt$

g8 : $ctlNeighbours = \varnothing$

g9 : $type(pkt) \neq DATA \wedge type(pkt) \neq RREQ$

g10 : $type(pkt) \neq RREQ \wedge type(pkt) \neq RREP$

g11 : $type(pkt) = RRER$

g12 : $pkt \mapsto eND \in pktErrND$

g13 : $y \mapsto eND \in fwdRouteTbl$

g14 : $y \mapsto eND \notin errND$

then

a1 : $updateNbrs := updateNbrs \setminus \{x \mapsto y\}$

a2 : $errND := errND \cup \{y \mapsto eND\}$

//node y records dead neighbour eND

end

Event *lose_all_neighbours* $\widehat{=}$

//(env) channel loses all neighbour

extends *lose_all_neighbours*

any

s

nbs

f

pkt

nxt

where

g1 : $channel \neq \varnothing$

g2 : $pkt \in ran(channel)$

g3 : $s = initialSrcAddr(pkt)$

g4 : $f \mapsto pkt \in channel$

g5 : $nbs = wsnLinks[\{f\}] \wedge nbs = \varnothing$

g6 : $pkt \in dom(netDestAddr) \wedge nxt = netDestAddr(pkt)$

g7 : $nxt = BROADCAST$

then

a1 : $envNeighbours := envNeighbours \cup \{pkt \mapsto FAILED\_XMIT\}$

end

Event *finish_tx_pkt* $\widehat{=}$

//(ctl) finishing packet transmission

extends *finish_tx_pkt*

any

pkt

f

where

g1 : $\{pkt\} \lhd ctlNeighbours = \varnothing$

g2 : $f \in ND \wedge f \mapsto pkt \in sentUp$

g3 : $pkt \in ran(sentUp) \wedge pkt \notin ran(sentDown) \wedge pkt \in ran(ndBuff)$

then

a1 : $sentUp := sentUp \setminus \{f \mapsto pkt\}$

end

Event *final_tx_pkt* $\widehat{=}$

//(ctl) the transmitted packet is removed from middleware indicating completion of transmission mechanism

extends *final_tx_pkt*

any

pkt

sk

where

g1 : $pkt \in PKT$

g2 : $pkt \in ran(sentUp) \wedge pkt \notin ran(sentDown) \wedge pkt \notin ran(ndBuff)$

g3 : $pkt \in dom(finalDestAddr) \wedge sk = finalDestAddr(pkt)$

g4 : $sk \in ND$

g5 : $pkt \in destBuff(sk)$

g6 : $\{pkt\} \lhd ctlNeighbours = \varnothing$

then

a1 : $sentUp := sentUp \rhd \{pkt\}$

end

Event *lose_specifc_neighbour* $\widehat{=}$

//(env) channel loses a specific neighbour

extends *lose_specifc_neighbour*

any

s

nbs

f

pkt

nxt

where

g1 : $channel \neq \varnothing$

g2 : $pkt \in ran(channel)$

g3 : $s = initialSrcAddr(pkt)$

g4 : $f \mapsto pkt \in channel$

g5 : $nbs = wsnLinks[\{f\}] \wedge nbs \neq \varnothing$

g6 : $pkt \in dom(netDestAddr) \wedge nxt = netDestAddr(pkt)$

g7 : $nxt \neq BROADCAST \wedge nxt \notin nbs$

then

a1 : $envNeighbours := envNeighbours \cup \{pkt \mapsto FAILED\_XMIT\}$

end

Event *lose_controlPkt* $\widehat{=}$

//(ctl) each node records a lost control packet

extends *lose_pkt*

any

f

pkt

where

g1 : $pkt \in ran(sentUp) \wedge pkt \notin ran(sentDown)$

g2 : $pkt \notin ran(recLostPkts)$

g3 : $pkt \notin ran(ndBuff)$

g4 : $f \mapsto pkt \in sentUp \wedge f \mapsto pkt \notin sentDown$

g5 : $f \mapsto pkt \notin recLostPkts$

g6 : $ran(\{pkt\} \lhd ctlNeighbours) = \{FAILED\_XMIT\}$

g7 : $type(pkt) \in CONTROL$

then

a1 : $sentUp := sentUp \rhd \{pkt\}$

a2 : $recLostPkts := recLostPkts \cup \{f \mapsto pkt\}$

a3 : $ctlNeighbours := ctlNeighbours \setminus \{pkt \mapsto FAILED\_XMIT\}$

end

Event *lose_dataPkt* $\widehat{=}$

//(ctl) each node records a lost data packet

extends *lose_pkt*

any

f

pkt

where

g1 : $pkt \in ran(sentUp) \wedge pkt \notin ran(sentDown)$

g2 : $pkt \notin ran(recLostPkts)$

g3 : $pkt \notin ran(ndBuff)$

g4 : $f \mapsto pkt \in sentUp \wedge f \mapsto pkt \notin sentDown$

g5 : $f \mapsto pkt \notin recLostPkts$

g6 : $ran(\{pkt\} \lhd ctlNeighbours) = \{FAILED\_XMIT\}$

g7 : $type(pkt) = DATA$

g8 : $f \mapsto pkt \notin rrerLists$

then

a1 : $sentUp := sentUp \rhd \{pkt\}$

a2 : $recLostPkts := recLostPkts \cup \{f \mapsto pkt\}$

a3 : $ctlNeighbours := ctlNeighbours \setminus \{pkt \mapsto FAILED\_XMIT\}$

a4 : $rrerLists := rrerLists \cup \{f \mapsto pkt\}$
//to initiate RRER reply

end

END

Appendix D

Event-B Patterns

D.1 Pattern Definition

D.1.1 *PSensingUnit*

D.1.1.1 Pattern *PSensingUnit*

Pattern PSensingUnit

Includes Context_PSensingUnit

Variables

$\$ctlSensedFlg\$$

$\$senseBuff\$$

Invariants

$\$ctlSensedFlg\$ \in ND \setminus \$Dests\$ \rightarrow BOOL \wedge$

$\$senseBuff\$ \in ND \setminus \$Dests\$ \rightarrow \mathbb{P}(\mathbb{Z})$

Initialisation

$\$ctlSensedFlg\$:= (ND \setminus \$Dests\$) \times \{FALSE\} \parallel$

$\$senseBuff\$:= (ND \setminus \$Dests\$) \times \{\varnothing\}$

Event *sensing* $\widehat{=}$

any

x

sf

$data$

where

$x \in ND \setminus \$Dests\$ \ \wedge \ sf \in BOOL \ \wedge \ data \in \mathbb{Z} \wedge$

$x \in dom(\$ctlSensedFlg\$) \wedge \$ctlSensedFlg\$(x) = FALSE$

then

$\$ctlSensedFlg\$(x) := sf \parallel$

$\$senseBuff\$(x) := \$senseBuff\$(x) \cup \{data\}$

end

Event $\$remove\_senseBuff\$ \mathrel{\widehat{=}}$

any

x

des

pkt

$data$

where

$x \in ND \setminus \$Dests\$ \ \wedge \ data \in \mathbb{Z} \wedge$

$\$senseBuff\$ \neq \varnothing \wedge data \in \$senseBuff\$(x)$

then

$\$senseBuff\$(x) := \$senseBuff\$(x) \setminus \{data\}$

end

END

D.1.1.2 Context *Context_PSensingUnit*

CONTEXT Context_PSensingUnit

SETS

PKT

$TYPE$

CONSTANTS

ND

$\$Dests\$$

$\$initialSrcAddr\$$

$type$

$DATA$

AXIOMS

$ND \subseteq \mathbb{N} \wedge$

$finite(ND) \wedge$

$\$Dests\$ \subseteq ND \ \wedge \ \wedge$

$finite(PKT) \wedge$

$\$initialSrcAddr\$ \in PKT \rightarrow ND \wedge$

$DATA \in TYPE \wedge$

$type \in PKT \rightarrow TYPE$

END

D.1.2 *PPacket*

D.1.2.1 Pattern *PPacket*

Pattern PPacket

Includes Context_PPacket

Variables

$\$pktFwdr\$$

$\$pktData\$$

$\$createdPkts\$$

Invariants

$\$pktFwdr\$ \in PKT \nrightarrow ND \wedge$

$\$pktData\$ \in PKT \nrightarrow \mathbb{Z} \wedge$

$\$createdPkts\$ \subseteq PKT$

Initialisation

$\$pktFwdr\$:= \varnothing \parallel$

$\$pktData\$:= \varnothing \parallel$

$\$createdPkts\$:= \varnothing$

Event $\$creating\_Pkt\$ \mathrel{\widehat{=}}$

any

x

des

pkt

$data$

where

$x \in ND \setminus \$Dests\$ \wedge pkt \in PKT \wedge data \in \mathbb{Z} \wedge$

$x = \$initialSrcAddr\$(pkt) \wedge des = ran(\{pkt\} \lhd \$finalDestAddr\$) \wedge$

$pkt \notin dom(\$pktFwdr\$) \wedge$

$pkt \notin dom(\$pktData\$)$

then

$\$createdPkts\$:= \$createdPkts\$ \cup \{pkt\} \parallel$

$\$pktFwdr\$:= \$pktFwdr\$ \cup \{pkt \mapsto x\} \parallel$

$\$pktData\$:= \$pktData\$ \cup \{pkt \mapsto data\}$

end

Event $\$set\_pktFwdr\$ \widehat{=}$

any

x

pkt

where

$x \in ND \wedge pkt \in PKT \wedge$

$pkt \in dom(\$pktFwdr\$)$

then

$\$pktFwdr\$:= \$pktFwdr\$ \lhd\!\!\!- \{pkt \mapsto x\}$

end

END

D.1.2.2 Context *Context_PPacket*

CONTEXT Context_PPacket

SETS

PKT

CONSTANTS

ND

$\$Dests\$$

$\$initialSrcAddr\$$

$\$finalDestAddr\$$

AXIOMS

$\$finalDestAddr\$ \in \mathbb{P}(PKT \times \mathbb{Z}) \wedge$

$ND \subseteq \mathbb{N} \wedge$

$finite(ND) \wedge$

$\$Dests\$ \subseteq ND \wedge$

$finite(PKT) \wedge$

$\$initialSrcAddr\$ \in PKT \rightarrow ND$

END

D.1.3 *PSend*

D.1.3.1 Pattern *PSend*

Pattern PSend

Includes Context_PSend

Variables

$\$sentDown\$$

$\$sentUp\$$

$\$ctlNeighbours\$$

Invariants

$\$sentUp\$ \in ND \leftrightarrow PKT \land$

$\$sentDown\$ \in ND \leftrightarrow PKT \land$

$\$ctlNeighbours\$ \in PKT \leftrightarrow ND$

Initialisation

$\$sentUp\$:= \varnothing \parallel$

$\$sentDown\$:= \varnothing \parallel$

$\$ctlNeighbours\$:= \varnothing$

Event $\$start\_tx\$ \mathrel{\widehat{=}}$

any

x

pkt

where

$x \in ND \land pkt \in PKT \land$

$x \mapsto pkt \notin \$sentDown\$$

then

$\$sentDown\$:= \$sentDown\$ \cup \{x \mapsto pkt\}$

end

Event $send\_down \mathrel{\widehat{=}}$

any

x

pkt

where

$x \in ND \land pkt \in PKT \land$

$x \mapsto pkt \in \$sentDown\$$

then

$skip$

end

Event $send\_up \mathrel{\widehat{=}}$

any

x

pkt

$nbrs$

where

$x \in ND \wedge pkt \in PKT \wedge$
$nbrs \in \{n | n \in \mathbb{P}(ND)\} \wedge nbrs \neq \varnothing \wedge$
$pkt \notin dom(\$ctlNeighbours\$) \wedge$
$x \mapsto pkt \notin \$sentUp\$ \wedge$
$x \mapsto pkt \in \$sentDown\$$

then

$\$ctlNeighbours\$:= \$ctlNeighbours\$ \cup (\{pkt\} \times nbrs) \parallel$
$\$sentDown\$:= \$sentDown\$ \setminus \{x \mapsto pkt\} \parallel$
$\$sentUp\$:= \$sentUp\$ \cup \{x \mapsto pkt\}$

end

Event $\$remove\_ctlNeighbours\$ \mathrel{\widehat{=}}$

any

x

pkt

where

$x \in ND \wedge pkt \in PKT \wedge$
$x \mapsto pkt \notin \$sentUp\$ \wedge$
$x \mapsto pkt \notin \$sentDown\$ \wedge$
$pkt \mapsto x \in \$ctlNeighbours\$$

then

$\$ctlNeighbours\$:= \$ctlNeighbours\$ \setminus \{pkt \mapsto nb\}$

end

Event $\$finish\_tx\_pkt\$ \mathrel{\widehat{=}}$

any

x

pkt

where

$x \in ND \wedge pkt \in PKT \wedge$
$\{pkt\} \lhd \$ctlNeighbours\$ = \varnothing \wedge$
$pkt \in ran(\$sentUp\$) \wedge pkt \notin ran(\$sentDown\$) \wedge$
$x \in dom(\$sentUp\$) \wedge$
$x \mapsto pkt \in \$sentUp\$$

then

$\$sentUp\$:= \$sentUp\$ \setminus \{x \mapsto pkt\}$

end

Event $\$final\_tx\_pkt\$ \mathrel{\widehat{=}}$

any

pkt

where

$pkt \in PKT \wedge$
$\{pkt\} \lhd \$ctlNeighbours\$ = \varnothing \wedge$
$pkt \in ran(\$sentUp\$) \wedge pkt \notin ran(\$sentDown\$)$

then

$\$sentUp\$:= \$sentUp\$ \rhd \{pkt\}$

end

END

D.1.3.2 Context *Context_PSend*

CONTEXT Context_PSend

SETS

PKT

CONSTANTS

ND

AXIOMS

$ND \subseteq \mathbb{N} \wedge$
$finite(ND) \wedge$
$finite(PKT)$

END

D.1.4 *PReceive*

D.1.4.1 Pattern *PReceive*

Pattern PReceive

Includes Context_PReceive

Variables

$\$recvBuff\$$
$\$clrRecvBuffFlg\$$

Invariants

$\$recvBuff\$ \in ND \leftrightarrow PKT \wedge$
$\$clrRecvBuffFlg\$ \in ND \leftrightarrow PKT \wedge$

Initialisation

$\$recvBuff\$:= \varnothing \parallel$

$\$clrRecvBuffFlg\$:= \varnothing$

Event $\$receive\$ \mathrel{\widehat{=}}$

any

x

pkt

where

$x \in ND \land pkt \in PKT \land$

$x \mapsto pkt \notin \$recvBuff\$$

then

$\$recvBuff\$:= \$recvBuff\$ \cup \{x \mapsto pkt\}$

end

Event $\$fwdr\_receive\_pkt\$ \mathrel{\widehat{=}}$

any

x

pkt

where

$x \in ND \land pkt \in PKT \land$

$x \mapsto pkt \in \$recvBuff\$ \land$

$x \mapsto pkt \notin \$clrRecvBuffFlg\$ \land$

$x \notin \$Dests\$$

then

$\$clrRecvBuffFlg\$:= \$clrRecvBuffFlg\$ \cup \{x \mapsto pkt\}$

end

Event $\$dest\_recv\_pkt\$ \mathrel{\widehat{=}}$

any

x

pkt

where

$x \in ND \land pkt \in PKT \land$

$x \mapsto pkt \in \$recvBuff\$ \land$

$x \in \$Dests\$ \land$

$x \mapsto pkt \notin \$clrRecvBuffFlg\$$

then

$\$clrRecvBuffFlg\$:= \$clrRecvBuffFlg\$ \cup \{nb \mapsto pkt\}$

end

Event $\$clear\_recvdBuff\$ \mathrel{\widehat{=}}$

any

x

pkt

where

$x \in ND \wedge pkt \in PKT \wedge$
$x \mapsto pkt \in \$recvBuff\$ \wedge$
$x \mapsto pkt \in \$clrRecvBuffFlg\$$

then

$\$recvBuff\$:= \$recvBuff\$ \setminus \{x \mapsto pkt\}$

end

END

D.1.4.2 Context *Context_PReceive*

CONTEXT Context_PReceive

SETS

PKT

CONSTANTS

ND

$\$Dests\$$

AXIOMS

$ND \subseteq \mathbb{N} \wedge$

$finite(ND) \wedge$

$\$Dests\$ \subseteq ND \wedge$

$finite(PKT)$

END

D.1.5 Pattern *PNDBuffer*

D.1.5.1 Pattern *PNDBuffer*

Pattern PNDBuffer

Includes Context

Variables

$\$ndBuff\$$

Invariants

$\$ndBuff\$ \in ND \leftrightarrow PKT$

Initialisation

$\$ndBuff\$:= \varnothing$

Event $\$record\_ndBuff\$ \mathrel{\widehat{=}}$

any

x

des

pkt

where

$x \in ND \setminus \$Dests\$ \wedge pkt \in PKT \wedge$
$x \mapsto pkt \notin \$ndBuff\$$

then

$\$ndBuff\$:= \$ndBuff\$ \cup \{x \mapsto pkt\}$

end

Event $\$remove\_ndBuff\$ \mathrel{\widehat{=}}$

any

x

pkt

where

$x \in ND \wedge pkt \in PKT \wedge$
$x \mapsto pkt \in \$ndBuff\$$

then

$\$ndBuff\$:= \$ndBuff\$ \setminus \{x \mapsto pkt\}$

end

Event $\$is\_In\_Range\_ndBuff\$ \mathrel{\widehat{=}}$

any

pkt

where

$pkt \in PKT \wedge pkt \in ran(\$ndBuff\$)$

then

$skip$

end

Event $\$isNot\_In\_Range\_ndBuff\$ \mathrel{\widehat{=}}$

any

pkt

where

$pkt \in PKT \wedge pkt \notin ran(\$ndBuff\$)$

then

$skip$

end

END

D.1.5.2 Context *Context_PwaitingBuffer*

CONTEXT Context_PwaitingBuffer

SETS

PKT

CONSTANTS

ND

$\$Dests\$$

$\$initialSrcAddr\$$

$\$finalDestAddr\$$

AXIOMS

$ND \subseteq \mathbb{N} \wedge$

$finite(ND) \wedge$

$\$Dests\$ \subseteq ND \wedge$

$finite(PKT) \wedge$

$\$initialSrcAddr\$ \in PKT \rightarrow ND$

END

D.1.6 *PDestBuffer*

D.1.6.1 Pattern *PDestBuffer*

Pattern PDestBuffer

Includes Context_IDestBuffer

Variables

$destBuff$

Invariants

$\$destBuff\$ \in \$Dests\$ \leftrightarrow PKT$

Initialisation

$\$destBuff\$:= \varnothing$

Event *$record_destBuff$* $\widehat{=}$

any

x

pkt

where

$x \in ND \wedge pkt \in PKT \wedge$

$x \in \$Dests\$ \wedge$
$x \mapsto pkt \notin \$destBuff\$$

then

$\$destBuff\$:= \$destBuff\$ \cup \{x \mapsto pkt\}$

end

Event *$remove_destBuff$* $\widehat{=}$

any

x
pkt

where

$x \in ND \wedge pkt \in PKT \wedge$
$x \in \$Dests\$ \wedge$
$x \mapsto pkt \in \$destBuff\$$

then

$\$destBuff\$:= \$destBuff\$ \setminus \{x \mapsto pkt\}$

end

Event *$is_In_Range_destBuff$* $\widehat{=}$

any

pkt

where

$pkt \in PKT \wedge pkt \in ran(\$destBuff\$)$

then

$skip$

end

Event *$isNot_In_Range_destBuff$* $\widehat{=}$

any

pkt

where

$pkt \in PKT \wedge pkt \notin ran(\$destBuff\$)$

then

$skip$

end

END

D.1.6.2 Context *Context_PDestBuffer*

CONTEXT Context_PDestBuffer

SETS

PKT

CONSTANTS

$\$Dests\$$

ND

AXIOMS

$ND \subseteq \mathbb{N} \wedge$

$finite(ND) \wedge$

$\$Dests\$ \subseteq ND \wedge$

$finite(PKT)$

END

D.2 Model Composition

D.2.1 *pM1*

MACHINE pM1

SEES cM1

VARIABLES

$pktFwdr$

$pktData$

$createdPkts$

$waitingBuff$

$sentDown$

$sentUp$

$ctlNeighbours$

$destBuff$

$recvBuff$

$clrRecvBuffFlg$

INVARIANTS

MPacket_inv_1 : $pktFwdr \in PKT \nrightarrow ND$

MPacket_inv_2 : $pktData \in PKT \nrightarrow \mathbb{Z}$

MPacket_inv_3 : $createdPkts \subseteq PKT$

MWaitingBuff_inv_1 : $waitingBuff \in ND \leftrightarrow PKT$

MSend_inv_1 : $sentUp \in ND \leftrightarrow PKT$

MSend_inv_2 : $sentDown \in ND \leftrightarrow PKT$

MSend_inv_3 : $ctlNeighbours \in PKT \leftrightarrow ND$

MDestBuffer_inv_1 : $destBuff \in Dests \leftrightarrow PKT$

MReceive_inv_1 : $recvBuff \in ND \leftrightarrow PKT$

MReceive_inv_2 : $clrRecvBuffFlg \in ND \leftrightarrow PKT$

EVENTS

Initialisation

begin

MPacket_int_1 : $pktFwdr := \varnothing$

MPacket_int_2 : $pktData := \varnothing$

MPacket_int_3 : $createdPkts := \varnothing$

MWaitingBuff_int_1 : $waitingBuff := \varnothing$

MSend_int_1 : $sentUp := \varnothing$

MSend_int_2 : $sentDown := \varnothing$

MSend_int_3 : $ctlNeighbours := \varnothing$

MDestBuffer_int_1 : $destBuff := \varnothing$

MReceive_int_1 : $recvBuff := \varnothing$

MReceive_int_2 : $clrRecvBuffFlg := \varnothing$

end

Event $creatingPkt \mathrel{\widehat{=}}$

any

x

des

pkt

$data$

where

MPacket_creating_pkt_g1 : $x \in ND \setminus Dests \wedge pkt \in PKT \wedge data \in \mathbb{Z}$

MPacket_creating_pkt_g2 : $x = initialSrc(pkt) \wedge des = ran(\{pkt\} \lhd finalDest)$

MPacket_creating_pkt_g3 : $pkt \notin dom(pktFwdr)$

MPacket_creating_pkt_g4 : $pkt \notin dom(pktData)$

MWaitingBuff_record_waitingBuff_g1 : $x \mapsto pkt \notin waitingBuff$

then

MPacket_creating_pkt_a1 : $createdPkts := createdPkts \cup \{pkt\}$

MPacket_creating_pkt_a2 : $pktFwdr := pktFwdr \cup \{pkt \mapsto x\}$

MPacket_creating_pkt_a3 : $pktData := pktData \cup \{pkt \mapsto data\}$

MWaitingBuff_record_waitingBuff_a1 : $waitingBuff := waitingBuff \cup \{x \mapsto pkt\}$

end

Event *start_tx* $\widehat{=}$

any

x

pkt

where

MSend_start_tx_g1 : $x \in ND \wedge pkt \in PKT$

MSend_start_tx_g2 : $x \mapsto pkt \notin sentDown$

MPacket_set_pktFwdr_g1 : $pkt \in dom(pktFwdr)$

MWaitingBuff_remove_waitingBuff_g1 : $x \mapsto pkt \in waitingBuff$

then

MISend_start_tx_a1 : $sentDown := sentDown \cup \{x \mapsto pkt\}$

MPacket_set_pktFwdr_a2 : $pktFwdr := pktFwdr \lhd\!\!\!+ \{pkt \mapsto x\}$

MWaitingBuff_remove_waitingBuff_a1 : $waitingBuff := waitingBuff \setminus \{x \mapsto pkt\}$

end

Event *send_down* $\widehat{=}$

any

x

pkt

where

MSend_send_down_g1 : $x \in ND \wedge pkt \in PKT$

MSend_send_down_g2 : $x \mapsto pkt \in sentDown$

then

$skip$

end

Event *send_up* $\widehat{=}$

any

x

pkt

$nbrs$

where

MSend_sent_up_g1 : $x \in ND \wedge pkt \in PKT$

MSend_sent_up_g2 : $x \mapsto pkt \notin sentUp$

MSend_sent_up_g3 : $x \mapsto pkt \in sentDown$

MSend_sent_up_g4 : $pkt \notin dom(ctlNeighbours)$

MSend_sent_up_g5 : $nbrs \in \{n | n \in \mathbb{P}(ND)\} \land nbrs \neq \varnothing$

then

MSend_sent_up_a1 : $ctlNeighbours := ctlNeighbours \cup (\{pkt\} \times nbrs)$

MSend_sent_up_a2 : $sentDown := sentDown \setminus \{x \mapsto pkt\}$

MSend_sent_up_a3 : $sentUp := sentUp \cup \{x \mapsto pkt\}$

end

Event *receive* $\widehat{=}$

any

x

pkt

where

MReceive_receive_g1 : $x \in ND \land pkt \in PKT$

MReceive_receive_g2 : $x \mapsto pkt \notin recvBuff$

MSend_remove_ctlNeighbour_g1 : $x \mapsto pkt \notin sentUp$

MSend_remove_ctlNeighbour_g2 : $x \mapsto pkt \notin sentDown$

MSend_remove_ctlNeighbour_g3 : $pkt \mapsto x \notin ctlNeighbours$

then

MReceive_receive_a1 : $recvBuff := recvBuff \cup \{x \mapsto pkt\}$

MSend_remove_ctlNeighbour_a1 : $ctlNeighbours := ctlNeighbours \setminus \{pkt \mapsto x\}$

end

Event *clear_recvdBuff* $\widehat{=}$

any

x

pkt

where

MReceive_clear_recvdBuff_g1 : $x \in ND \land pkt \in PKT$

MReceive_clear_recvdBuff_g2 : $x \mapsto pkt \in recvBuff$

MReceive_clear_recvdBuff_g3 : $x \mapsto pkt \in clrRecvBuffFlg$

then

MReceive_clear_recvdBuff_a1 : $recvBuff := recvBuff \setminus \{x \mapsto pkt\}$

end

Event *fwdr_receive_pkt* $\widehat{=}$

any

x

pkt

where

MReceive_fwdr_receive_pkt_g1 : $x \in ND \wedge pkt \in PKT$

MReceive_fwdr_receive_pkt_g2 : $x \mapsto pkt \in recvBuff$

MReceive_fwdr_receive_pkt_g3 : $x \mapsto pkt \notin clrRecvBuffFlg$

MReceive_fwdr_receive_pkt_g4 : $x \notin Dests$

MWaitingBuff_record_waitingBuff_g1 : $x \mapsto pkt \notin waitingBuff$

then

MReceive_fwdr_receive_pkt_a1 : $clrRecvBuffFlg := clrRecvBuffFlg \cup \{nb \mapsto pkt\}$

MWaitingBuff_record_waitingBuff_a1 : $waitingBuff := waitingBuff \cup \{nb \mapsto pkt\}$

end

Event *dest_recv_pkt* $\widehat{=}$

any

x

pkt

where

MReceive_dest_receive_pkt_g1 : $x \in ND \wedge pkt \in PKT$

MReceive_dest_receive_pkt_g2 : $x \mapsto pkt \in recvBuff$

MReceive_dest_receive_pkt_g3 : $x \in Dests$

MReceive_dest_receive_pkt_g4 : $x \mapsto pkt \notin clrRecvBuffFlg$

MDestBuffer_record_destBuff_g1 : $x \mapsto pkt \notin destBuff$

then

MDestBuffer_waitingBuff_a1 : $destBuff := destBuff \cup \{x \mapsto pkt\}$

MDestBuffer_record_destBuff_a1 : $clrRecvBuffFlg := clrRecvBuffFlg \cup \{x \mapsto pkt\}$

end

Event *finish_tx_pkt* $\widehat{=}$

any

x

pkt

where

MSend_finish_tx_pkt_g1 : $x \in ND \wedge pkt \in PKT$

MSend_finish_tx_pkt_g2 : $\{pkt\} \lhd ctlNeighbours = \varnothing$

MSend_finish_tx_pkt_g3 : $pkt \in ran(sentUp) \wedge pkt \notin ran(sentDown)$

MSend_finish_tx_pkt_g4 : $x \in dom(sentUp)$

MSend_finish_tx_pkt_g5 : $x \mapsto pkt \in sentUp$

MwaitingBuffer_is_In_Range_waitingBuff_g1 : $pkt \in ran(waitingBuff)$

then

MSend_finish_tx_pkt_a1 : $sentUp := sentUp \setminus \{x \mapsto pkt\}$

end

Event *final_tx_pkt* $\widehat{=}$

any

pkt

where

MSend_final_tx_pkt_g1 : $pkt \in PKT$

MSend_final_tx_pkt_g2 : $\{pkt\} \lhd ctlNeighbours = \varnothing$

MSend_final_tx_pkt_g3 : $pkt \in ran(sentUp) \wedge pkt \notin ran(sentDown)$

MwaitingBuffer_isNot_In_Range_waitingBuff_g1 : $pkt \notin ran(waitingBuff)$

then

MSend_final_tx_pkt_a1 : $sentUp := sentUp \rhd \{pkt\}$

end

END

D.2.2 *uM2*

MACHINE uM2

REFINES pM1

SEES cM2

VARIABLES

$pktFwdr$

$pktData$

$createdPkts$

$waitingBuff$

$sentDown$

$sentUp$

$ctlNeighbours$

$destBuff$

$recvBuff$

$clrRecvBuffFlg$

EVENTS

Initialisation

extended

begin

MPacket_int_1 : $pktFwdr := \varnothing$

MPacket_int_2 : $pktData := \varnothing$

MPacket_int_3 : $createdPkts := \varnothing$

MwaitingBuff_int_1 : $waitingBuff := \varnothing$

MSend_int_1 : $sentUp := \varnothing$

MSend_int_2 : $sentDown := \varnothing$

MSend_int_3 : $ctlNeighbours := \varnothing$

MDestBuffer_int_1 : $destBuff := \varnothing$

MReceive_int_1 : $recvBuff := \varnothing$

MReceive_int_2 : $clrRecvBuffFlg := \varnothing$

end

Event $creatingDataPacket \mathrel{\widehat{=}}$

refines $creatingPkt$

any

x

des

pkt

$data$

where

MPacket_creating_pkt_g1 : $x \in ND \setminus Dests \wedge pkt \in PKT \wedge data \in \mathbb{Z}$

MPacket_creating_pkt_g2 : $x = initialSrc(pkt) \wedge des = ran(\{pkt\} \lhd finalDest)$

MPacket_creating_pkt_g3 : $pkt \notin dom(pktFwdr)$

MPacket_creating_pkt_g4 : $pkt \notin dom(pktData)$

MwaitingBuffMgt_record_waitingBuff_g1 : $x \mapsto pkt \notin waitingBuff$

User_defined_guard_g1 : $type(pkt) = DATA$

then

MPacket_creating_pkt_a1 : $createdPkts := createdPkts \cup \{pkt\}$

MPacket_creating_pkt_a2 : $pktFwdr := pktFwdr \cup \{pkt \mapsto x\}$

MPacket_creating_pkt_a3 : $pktData := pktData \cup \{pkt \mapsto data\}$

MwaitingBuffMgt_record_waitingBuff_a1 : $waitingBuff := waitingBuff \cup \{x \mapsto pkt\}$

end

Event *creatingControlPacket* $\widehat{=}$

refines *creatingPkt*

any

x

des

pkt

$data$

where

MPacket_creating_pkt_g1 : $x \in ND \setminus Dests \wedge pkt \in PKT \wedge data \in \mathbb{Z}$

MPacket_creating_pkt_g2 : $x = initialSrc(pkt) \wedge des = ran(\{pkt\} \lhd finalDest)$

MPacket_creating_pkt_g3 : $pkt \notin dom(pktFwdr)$

MPacket_creating_pkt_g4 : $pkt \notin dom(pktData)$

MwaitingBuff_record_waitingBuff_g1 : $x \mapsto pkt \notin waitingBuff$

User_defined_guard_g1 : $data = CTL\_VAL$

User_defined_guard_g2 : $type(pkt) \in CONTROL$

then

MPacket_creating_pkt_a1 : $createdPkts := createdPkts \cup \{pkt\}$

MPacket_creating_pkt_a2 : $pktFwdr := pktFwdr \cup \{pkt \mapsto x\}$

MPacket_creating_pkt_a3 : $pktData := pktData \cup \{pkt \mapsto data\}$

MWaitingBuff_record_waitingBuff_a1 : $waitingBuff := waitingBuff \cup \{x \mapsto pkt\}$

end

Event *start_tx* $\widehat{=}$

extends *start_tx*

any

x

pkt

where

MSend_start_tx_g1 : $x \in ND \wedge pkt \in PKT$

MSend_start_tx_g2 : $x \mapsto pkt \notin sentDown$

MPacket_set_pktFwdr_g1 : $pkt \in dom(pktFwdr)$

MWaitingBuff_remove_waitingBuff_g1 : $x \mapsto pkt \in waitingBuff$

then

MISend_start_tx_a1 : $sentDown := sentDown \cup \{x \mapsto pkt\}$

MPacket_set_pktFwdr_a2 : $pktFwdr := pktFwdr \lhd\!\!\!- \{pkt \mapsto x\}$

MWaitingBuff_remove_waitingBuff_a1 : $waitingBuff := waitingBuff \setminus \{x \mapsto pkt\}$

end

Event *send_down* $\widehat{=}$

extends *send_down*

any

x

pkt

where

MSend_send_down_g1 : $x \in ND \wedge pkt \in PKT$

MSend_send_down_g2 : $x \mapsto pkt \in sentDown$

then

$skip$

end

Event *send_up* $\widehat{=}$

extends *send_up*

any

x

pkt

$nbrs$

where

MSend_sent_up_g1 : $x \in ND \wedge pkt \in PKT$

MSend_sent_up_g2 : $x \mapsto pkt \notin sentUp$

MSend_sent_up_g3 : $x \mapsto pkt \in sentDown$

MSend_sent_up_g4 : $pkt \notin dom(ctlNeighbours)$

MSend_sent_up_g5 : $nbrs \in \{n | n \in \mathbb{P}(ND)\} \wedge nbrs \neq \varnothing$

then

MSend_sent_up_a1 : $ctlNeighbours := ctlNeighbours \cup (\{pkt\} \times nbrs)$

MSend_sent_up_a2 : $sentDown := sentDown \setminus \{x \mapsto pkt\}$

MSend_sent_up_a3 : $sentUp := sentUp \cup \{x \mapsto pkt\}$

end

Event *receive* $\widehat{=}$

extends *receive*

any

x

pkt

where

MReceive_receive_g1 : $x \in ND \wedge pkt \in PKT$

MReceive_receive_g2 : $x \mapsto pkt \notin recvBuff$

MSend_remove_ctlNeighbour_g1 : $x \mapsto pkt \notin sentUp$

MSend_remove_ctlNeighbour_g2 : $x \mapsto pkt \notin sentDown$

MSend_remove_ctlNeighbour_g3 : $pkt \mapsto x \notin ctlNeighbours$

then

MReceive_receive_a1 : $recvBuff := recvBuff \cup \{x \mapsto pkt\}$

MSend_remove_ctlNeighbour_a1 : $ctlNeighbours := ctlNeighbours \setminus \{pkt \mapsto x\}$

end

Event *clear_recvdBuff* $\widehat{=}$

extends *clear_recvdBuff*

any

x

pkt

where

MReceive_clear_recvdBuff_g1 : $x \in ND \wedge pkt \in PKT$

MReceive_clear_recvdBuff_g2 : $x \mapsto pkt \in recvBuff$

MReceive_clear_recvdBuff_g3 : $x \mapsto pkt \in clrRecvBuffFlg$

then

MReceive_clear_recvdBuff_a1 : $recvBuff := recvBuff \setminus \{x \mapsto pkt\}$

end

Event *fwdr_receive_pkt* $\widehat{=}$

extends *fwdr_receive_pkt*

any

x

pkt

where

MReceive_fwdr_receive_pkt_g1 : $x \in ND \wedge pkt \in PKT$

MReceive_fwdr_receive_pkt_g2 : $x \mapsto pkt \in recvBuff$

MReceive_fwdr_receive_pkt_g3 : $x \mapsto pkt \notin clrRecvBuffFlg$

MReceive_fwdr_receive_pkt_g4 : $x \notin Dests$

MWaitingBuff_record_waitingBuff_g1 : $x \mapsto pkt \notin waitingBuff$

then

MReceive_fwdr_receive_pkt_a1 : $clrRecvBuffFlg := clrRecvBuffFlg \cup \{x \mapsto pkt\}$

MwaitingBuffer_record_waitingBuff_a1 : $waitingBuff := waitingBuff \cup \{x \mapsto pkt\}$

end

Event *dest_recv_pkt* $\widehat{=}$

extends *dest_recv_pkt*

any

nb

pkt

where

`MReceive_dest_receive_pkt_g1` : $nb \in ND \wedge pkt \in PKT$

`MReceive_dest_receive_pkt_g2` : $nb \mapsto pkt \in recvBuff$

`MReceive_dest_receive_pkt_g3` : $nb \in Dests$

`MReceive_dest_receive_pkt_g4` : $nb \mapsto pkt \notin clrRecvBuffFlg$

`MDestBuffer_record_destBuff_g1` : $nb \mapsto pkt \notin destBuff$

then

`MDestBuffer_waitingBuff_a1` : $destBuff := destBuff \cup \{nb \mapsto pkt\}$

`MDestBuffer_record_destBuff_a1` : $clrRecvBuffFlg := clrRecvBuffFlg \cup \{nb \mapsto pkt\}$

end

Event *finish_tx_pkt* $\widehat{=}$

extends *finish_tx_pkt*

any

x

pkt

where

`MSend_finish_tx_pkt_g1` : $x \in ND \wedge pkt \in PKT$

`MSend_finish_tx_pkt_g2` : $\{pkt\} \lhd ctlNeighbours = \varnothing$

`MSend_finish_tx_pkt_g3` : $pkt \in ran(sentUp) \wedge pkt \notin ran(sentDown)$

`MSend_finish_tx_pkt_g4` : $x \in dom(sentUp)$

`MSend_finish_tx_pkt_g5` : $x \mapsto pkt \in sentUp$

then

`MSend_finish_tx_pkt_a1` : $sentUp := sentUp \setminus \{x \mapsto pkt\}$

end

END

D.2.3 *pM3*

MACHINE pM3

REFINES uM2

SEES cM2

VARIABLES

$pktFwdr$

$pktData$

$createdPkts$

$waitingBuff$

$sentDown$

$sentUp$

$ctlNeighbours$

$destBuff$

$recvBuff$

$clrRecvBuffFlg$

$ctlSensedFlg$

$senseBuff$

INVARIANTS

MSensingUnit_sensing_inv_1 : $ctlSensedFlg \in ND \setminus Dests \rightarrow BOOL$

MSensingUnit_sensing_inv_2 : $senseBuff \in ND \setminus Dests \rightarrow \mathbb{P}(\mathbb{Z})$

EVENTS

Initialisation

extended

begin

MPacket_int_1 : $pktFwdr := \varnothing$

MPacket_int_2 : $pktData := \varnothing$

MPacket_int_3 : $createdPkts := \varnothing$

MwaitingBuffer_int_1 : $waitingBuff := \varnothing$

MSend_int_1 : $sentUp := \varnothing$

MSend_int_2 : $sentDown := \varnothing$

MSend_int_3 : $ctlNeighbours := \varnothing$

MDestBuffer_int_1 : $destBuff := \varnothing$

MReceive_int_1 : $recvBuff := \varnothing$

MReceive_int_2 : $clrRecvBuffFlg := \varnothing$

MSensingUnit_sensing_int_1 : $ctlSensedFlg := (ND \setminus Dests) \times \{FALSE\}$

MSensingUnit_sensing_int_2 : $senseBuff := (ND \setminus Dests) \times \{\varnothing\}$

end

Event *sensing* $\widehat{=}$

any

x

sf

$data$

where

MSensingUnit_sensing_g1 : $x \in ND \setminus Dests$

MSensingUnit_sensing_g2 : $sf \in BOOL$

MSensingUnit_sensing_g3 : $data \in \mathbb{Z}$

MSensingUnit_sensing_g4 : $x \in dom(ctlSensedFlg) \wedge ctlSensedFlg(x) = FALSE$

then

MSensingUnit_sensing_a1 : $ctlSensedFlg(x) := sf$

MSensingUnit_sensing_a2 : $senseBuff(x) := senseBuff(x) \cup \{data\}$

end

Event *creatingDataPacket* $\widehat{=}$

extends *creatingDataPacket*

any

x

des

pkt

$data$

where

MPacket_creating_pkt_g1 : $x \in ND \setminus Dests \wedge pkt \in PKT \wedge data \in \mathbb{Z}$

MPacket_creating_pkt_g2 : $x = initialSrc(pkt) \wedge des = ran(\{pkt\} \lhd finalDest)$

MPacket_creating_pkt_g3 : $pkt \notin dom(pktFwdr)$

MPacket_creating_pkt_g4 : $pkt \notin dom(pktData)$

MwaitingBuffMgt_record_waitingBuff_g1 : $x \mapsto pkt \notin waitingBuff$

User_defined_guard_g1 : $type(pkt) = DATA$

MSensingUnit_removeSenseBuff_g1 : $senseBuff \neq \varnothing \wedge data \in senseBuff(x)$

then

MPacket_creating_pkt_a1 : $createdPkts := createdPkts \cup \{pkt\}$

MPacket_creating_pkt_a2 : $pktFwdr := pktFwdr \cup \{pkt \mapsto x\}$

MPacket_creating_pkt_a3 : $pktData := pktData \cup \{pkt \mapsto data\}$

MwaitingBuffMgt_record_waitingBuff_a1 : $waitingBuff := waitingBuff \cup \{x \mapsto pkt\}$

MSensingUnit_removeSenseBuff_a1 : $senseBuff(x) := senseBuff(x) \setminus \{data\}$

end

Event *creatingControlPacket* $\widehat{=}$

extends *creatingControlPacket*

any

x

des

pkt

$data$

where

MPacket_creating_pkt_g1 : $x \in ND \setminus Dests \wedge pkt \in PKT \wedge data \in \mathbb{Z}$

MPacket_creating_pkt_g2 : $x = initialSrc(pkt) \wedge des = ran(\{pkt\} \lhd finalDest)$

MPacket_creating_pkt_g3 : $pkt \notin dom(pktFwdr)$

MPacket_creating_pkt_g4 : $pkt \notin dom(pktData)$

MwaitingBuffMgt_record_waitingBuff_g1 : $x \mapsto pkt \notin waitingBuff$

User_defined_guard_g1 : $data = CTL\_VAL$

User_defined_guard_g2 : $type(pkt) \in CONTROL$

then

MPacket_creating_pkt_a1 : $createdPkts := createdPkts \cup \{pkt\}$

MPacket_creating_pkt_a2 : $pktFwdr := pktFwdr \cup \{pkt \mapsto x\}$

MPacket_creating_pkt_a3 : $pktData := pktData \cup \{pkt \mapsto data\}$

MwaitingBuffMgt_record_waitingBuff_a1 : $waitingBuff := waitingBuff \cup \{x \mapsto pkt\}$

end

Event *start_tx* $\widehat{=}$

extends *start_tx*

any

x

pkt

where

MSend_start_tx_g1 : $x \in ND \wedge pkt \in PKT$

MSend_start_tx_g2 : $x \mapsto pkt \notin sentDown$

MPacket_set_pktFwdr_g1 : $pkt \in dom(pktFwdr)$

MwaitingBuffer_remove_waitingBuff_g1 : $x \mapsto pkt \in waitingBuff$

then

MISend_start_tx_a1 : $sentDown := sentDown \cup \{x \mapsto pkt\}$

MPacket_set_pktFwdr_a2 : $pktFwdr := pktFwdr \lhd\!\!+ \{pkt \mapsto x\}$

MwaitingBuffer_remove_waitingBuff_a1 : $waitingBuff := waitingBuff \setminus \{x \mapsto pkt\}$

end

Event *send_down* $\widehat{=}$

extends *send_down*

any

x

pkt

where

MSend_send_down_g1 : $x \in ND \land pkt \in PKT$

MSend_send_down_g2 : $x \mapsto pkt \in sentDown$

then

$skip$

end

Event *send_up* $\widehat{=}$

extends *send_up*

any

x

pkt

$nbrs$

where

MSend_sent_up_g1 : $x \in ND \land pkt \in PKT$

MSend_sent_up_g2 : $x \mapsto pkt \notin sentUp$

MSend_sent_up_g3 : $x \mapsto pkt \in sentDown$

MSend_sent_up_g4 : $pkt \notin dom(ctlNeighbours)$

MSend_sent_up_g5 : $nbrs \in \{n | n \in \mathbb{P}(ND)\} \land nbrs \neq \varnothing$

then

MSend_sent_up_a1 : $ctlNeighbours := ctlNeighbours \cup (\{pkt\} \times nbrs)$

MSend_sent_up_a2 : $sentDown := sentDown \setminus \{x \mapsto pkt\}$

MSend_sent_up_a3 : $sentUp := sentUp \cup \{x \mapsto pkt\}$

end

Event *receive* $\widehat{=}$

extends *receive*

any

x

pkt

where

MReceive_receive_g1 : $x \in ND \land pkt \in PKT$

MReceive_receive_g2 : $x \mapsto pkt \notin recvBuff$

MSend_remove_ctlNeighbour_g1 : $x \mapsto pkt \notin sentUp$

MSend_remove_ctlNeighbour_g2 : $x \mapsto pkt \notin sentDown$

MSend_remove_ctlNeighbour_g3 : $pkt \mapsto x \notin ctlNeighbours$

then

MReceive_receive_a1 : $recvBuff := recvBuff \cup \{x \mapsto pkt\}$

MSend_remove_ctlNeighbour_a1 : $ctlNeighbours := ctlNeighbours \setminus \{pkt \mapsto x\}$

end

Event *clear_recvdBuff* $\widehat{=}$

extends *clear_recvdBuff*

any

x

pkt

where

MReceive_clear_recvdBuff_g1 : $x \in ND \land pkt \in PKT$

MReceive_clear_recvdBuff_g2 : $x \mapsto pkt \in recvBuff$

MReceive_clear_recvdBuff_g3 : $x \mapsto pkt \in clrRecvBuffFlg$

then

MReceive_clear_recvdBuff_a1 : $recvBuff := recvBuff \setminus \{x \mapsto pkt\}$

end

Event *fwdr_receive_pkt* $\widehat{=}$

extends *fwdr_receive_pkt*

any

x

pkt

where

MReceive_fwdr_receive_pkt_g1 : $x \in ND \land pkt \in PKT$

MReceive_fwdr_receive_pkt_g2 : $x \mapsto pkt \in recvBuff$

MReceive_fwdr_receive_pkt_g3 : $x \mapsto pkt \notin clrRecvBuffFlg$

MReceive_fwdr_receive_pkt_g4 : $x \notin Dests$

MwaitingBuffer_record_waitingBuff_g1 : $x \mapsto pkt \notin waitingBuff$

then

MReceive_fwdr_receive_pkt_a1 : $clrRecvBuffFlg := clrRecvBuffFlg \cup \{x \mapsto pkt\}$

MwaitingBuffer_record_waitingBuff_a1 : $waitingBuff := waitingBuff \cup \{x \mapsto pkt\}$

end

Event *dest_recv_pkt* $\widehat{=}$

extends *dest_recv_pkt*

any

x

pkt

where

MReceive_dest_receive_pkt_g1 : $x \in ND \wedge pkt \in PKT$

MReceive_dest_receive_pkt_g2 : $x \mapsto pkt \in recvBuff$

MReceive_dest_receive_pkt_g3 : $x \in Dests$

MReceive_dest_receive_pkt_g4 : $x \mapsto pkt \notin clrRecvBuffFlg$

MDestBuffer_record_destBuff_g1 : $x \mapsto pkt \notin destBuff$

then

MDestBuffer_waitingBuff_a1 : $destBuff := destBuff \cup \{x \mapsto pkt\}$

MDestBuffer_record_destBuff_a1 : $clrRecvBuffFlg := clrRecvBuffFlg \cup \{x \mapsto pkt\}$

end

Event *finish_tx_pkt* $\widehat{=}$

extends *finish_tx_pkt*

any

x

pkt

where

MSend_finish_tx_pkt_g1 : $x \in ND \wedge pkt \in PKT$

MSend_finish_tx_pkt_g2 : $\{pkt\} \lhd ctlNeighbours = \varnothing$

MSend_finish_tx_pkt_g3 : $pkt \in ran(sentUp) \wedge pkt \notin ran(sentDown)$

MSend_finish_tx_pkt_g4 : $x \in dom(sentUp)$

MSend_finish_tx_pkt_g5 : $x \mapsto pkt \in sentUp$

then

MSend_finish_tx_pkt_a1 : $sentUp := sentUp \setminus \{x \mapsto pkt\}$

end

Event *final_tx_pkt* $\widehat{=}$

extends *final_tx_pkt*

any

pkt

where

MSend_final_tx_pkt_g1 : $pkt \in PKT$

MSend_final_tx_pkt_g2 : $\{pkt\} \lhd ctlNeighbours = \varnothing$

MSend_final_tx_pkt_g3 : $pkt \in ran(sentUp) \wedge pkt \notin ran(sentDown)$

MWaitingBuff_isNot_In_Range_waitingBuff_g1 : $pkt \notin ran(waitingBuff)$

then

MSend_final_tx_pkt_a1 : $sentUp := sentUp \rhd \{pkt\}$

end

END

References

(2004). *Model-checking large network protocol implementations*. USENIX.

(2008). Wiki. ProB [online]. http://wiki.event-b.org/index.php/ProB.

(2011a). Unified Modeling Language [online]. http://www.uml.org.

(2011b). Wiki. UML-B [online]. http://wiki.event-b.org/index.php/UML-B.

Abrial, J., Cansell, D., and Méry, D. (2003). A mechanically proved and incremental development of IEEE 1394 tree identify protocol. *Formal Asp. Comput.*, 14(3):215–227.

Abrial, J.-R. (1996). *The B-book: assigning programs to meanings*. Cambridge University Press, New York, NY, USA.

Abrial, J.-R. (2007). Formal Methods : Theory Becoming Practice. *Journal of Universal Computer Science*, 13(5):619–628.

Abrial, J.-R. (2010). *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, New York, NY, USA, 1st edition.

Abrial, J.-R., Butler, M., and Hallerstede, S. (2006). An open extensible tool environment for Event-B. In *ICFEM 2006*, November 2006, Macau.

Abrial, J.-R. and Hallerstede, S. (2007). Refinement, decomposition, and instantiation of discrete models: Application to event-b. *Fundam. Inf.*, 77(1-2):1–28.

ADVANCE (2013). Advance deliverable d4.2 (issue 2): Methods and tools for simulation and testing I. Technical report.

Akkaya, K. and Younis, M. F. (2005). A survey on routing protocols for wireless sensor networks. *Ad Hoc Networks*, 3(3):325–349.

Akyildiz, I. F., Su, W., Sankarasubramaniam, Y., and Cayirci, E. (2002). A survey on sensor networks. *IEEE Communications Magazine*, 40(8):102 – 114. Survey.

Al-Karaki, J. N. and Kamal, A. E. (2004). Routing techniques in wireless sensor networks: A survey. *Wireless Commun.*, 11(6):6–28.

Alkhammash, E., Fathabadi, A. S., Butler, M. J., and Cîrstea, C. (2013). Building traceable event-b models from requirements. *ECEASST*, 66.

Allen, M., Challen, G., and Brusey, J. (2010). Designing for deployment. In Guara, E., Girod, L., Brusey, J., Allen, M., and Challen, G., editors, *Wireless Sensor Networks: Deployments and Design Frameworks*. Springer.

Altman, E. and Jiménez, T. (2012). *NS Simulator for Beginners*. Synthesis Lectures on Communication Networks. Morgan & Claypool Publishers.

Arora, A., Dutta, P., Bapat, S., Kulathumani, V., Zhang, H., Naik, V., Mittal, V., Cao, H., Demirbas, M., Gouda, M., Choi, Y., Herman, T., Kulkarni, S., Arumugam, U., Nesterenko, M., Vora, A., and Miyashita, M. (2004). A line in the sand: A wireless sensor network for target detection, classification, and tracking. *Computer Networks (Elsevier*, 46:605–634.

Back, R. and von Wright, J. (1990). Refinement calculus, part i: Sequential nondeterministic programs. In de Bakker, J., de Roever, W.-P., and Rozenberg, G., editors, *Stepwise Refinement of Distributed Systems Models, Formalisms, Correctness*, volume 430 of *Lecture Notes in Computer Science*, pages 42–66. Springer Berlin Heidelberg.

Back, R. J. R. (1980). Correctness preserving program refinements: Proof theory and applications. *Mathematical Center Tracts*, 131.

Back, R. J. R. (1990). Refinement calculus, part ii: Parallel and reactive programs. In *Proceedings on Stepwise Refinement of Distributed Systems: Models, Formalisms, Correctness*, REX workshop, pages 67–93, New York, NY, USA. Springer-Verlag New York, Inc.

Back, R. J. R. and Kurki-Suonio, F. (1988). Distributed cooperation with action systems. *ACM Trans. Program. Lang. Syst.*, 10(4):513–554.

Baldus, H., Klabunde, K., and M?, G. (2004). Reliable set-up of medical body-sensor networks. In Karl, H., Willig, A., and Wolisz, A., editors, *EWSN*, volume 2920 of *Lecture Notes in Computer Science*, pages 353–363. Springer.

Banker, R., Kauffman, R., and Zweig, D. (1993). Repository evaluation of software reuse. *IEEE Transactions on Software Engineering*, 19(4):379–389.

Barrenetxea, G., Ingelrest, F., Schaefer, G., and Vetterli, M. (2008a). The hitchhiker's guide to successful wireless sensor network deployments. In *Proceedings of the 6th ACM conference on Embedded network sensor systems*, SenSys '08, pages 43–56, New York, NY, USA. ACM.

Barrenetxea, G., Ingelrest, F., Schaefer, G., Vetterli, M., Couach, O., and Parlange, M. (2008b). Sensorscope: Out-of-the-box environmental monitoring. In *IPSN*, pages 332–343.

Behm, P., Benoit, P., Faivre, A., and Meynadier, J.-M. (1999). Mtor: A successful application of b in a large project. In Wing, J., Woodcock, J., and Davies, J., editors, *FM?99 Formal Methods*, volume 1708 of *Lecture Notes in Computer Science*, pages 369–387. Springer Berlin Heidelberg.

Behrmann, G., David, A., and Larsen, K. G. (2004). A tutorial on uppaal. In Bernardo, M. and Corradini, F., editors, *Formal Methods for the Design of Real-Time Systems, International School on Formal Methods for the Design of Computer, Communication and Software Systems, SFM-RT 2004, Bertinoro, Italy, September 13-18, 2004, Revised Lectures*, volume 3185 of *Lecture Notes in Computer Science*, pages 200–236. Springer.

Bendiposto, J., Clark, J., Dobrikov, I., Körner, P., Krings, S., Ladenberger, L., Leuschel, M., and Plagge, D. (2013). Prob 2.0 tutorial. In *Proceedings of the 4th Rodin User and Developer Workshop*, TUCS Lecture Notes. TUCS.

Bendiposto, J., Clark, J., and Körner, P. (2012). Prob 2.0 core api. https://github.com/bendisposto/probcore.

Bendisposto, J. and Clark, J. (2014). Prob 2.0 developer handbook. http://nightly.cobra.cs.uni-duesseldorf.de/prob2/prob2-handbook/nightly/devel/html/index.html.

Bernardeschi, C. and Masci, P. (2008). Early prototyping of wireless sensor network algorithms in pvs. In *Proceedings of the 27th international conference on Computer Safety, Reliability, and Security*, pages 346–359, Newcastle upon Tyne, UK. Springer-Verlag.

Beutel, J., Roemer, K., Woehrle, M., and Ringwald, M. (2010). Deployment techniques for sensor networks. In *Sensor Networks - Where Theory Meets Practice*, pages 219–248. Springer.

Bhargavan, K., Obradovic, D., and Gunter, C. A. (2002). Formal verification of standards for distance vector routing protocols. *J. ACM*, 49(4):538–576.

Bhatti, S., Xu, J., and Memon, M. (2010). Model checking of a target tracking protocol for wireless sensor networks. In *CIT*, pages 2867–2872. IEEE Computer Society.

Bicarregui, J. and Ritchie, B. (1995). Invariants, frames, and postconditions: A comparison of the vdm and b notations. *IEEE Trans. Software Eng.*, 21(2):79–89.

Bloomfield, R. E., Craigen, D., Koob, F., Ullmann, M., and Wittmann, S. (2000). Formal methods diffusion: Past lessons and future prospects. In Koornneef, F. and van der Meulen, M., editors, *SAFECOMP*, volume 1943 of *Lecture Notes in Computer Science*, pages 211–226. Springer.

Boulis, A. (2011). Castalia: A simulator for Wireless Sensor Networks and Body Area Networks [online]. http://castalia.npc.nicta.com.au/documentation.php.

Boulis, A., Fehnker, A., Fruth, M., and McIver, A. (2008). Cavi – simulation and model checking for wireless sensor networks. In *QEST '08: Proceedings of the 2008 Fifth International Conference on Quantitative Evaluation of Systems*, pages 37–38, Washington, DC, USA. IEEE Computer Society.

Bowen, J. (2003). *Formal specification and documentation using z: A case study approach*. Thomson Publishing.

Broenink, J. F., Kleijn, C., Larsen, P. G., Jovanovic, D., Verhoef, M., and Pierce, K. (2010). Design support and tooling for dependable embedded control software. In *Proceedings of the 2Nd International Workshop on Software Engineering for Resilient Systems*, SERENE '10, pages 77–82, NY, USA. ACM.

Butler, M. (1992). *A CSP Approach to Action Systems*. PhD thesis, Oxford University, Computing Laboratory.

Butler, M. (2007). The rodin formal modelling tool. In *Christmas 2007 Meeting-Formal*, pages 1–5, London.

Butler, M. (2008). Incremental Design of Distributed Systems with Event-B. In *Engineering Methods and Tools for Software Safety and Security - Marktoberdorf Summer School 2008*, pages 131–160. IOS Press.

Butler, M. (2009). Decomposition structures for event-b. In *Integrated Formal Methods*, pages 20–38, Berlin, Heidelberg. Springer-Verlag.

Canli, T., Hefeida, M., and Khokhar, A. (2010). Bulkmac: a cross-layer based mac protocol for wireless sensor networks. In *Proceedings of the 6th International Wireless Communications and Mobile Computing Conference*, IWCMC '10, pages 442–446, New York, NY, USA. ACM.

Cardell-Oliver, R. (1991). On the use of the hol system for protocol verification. In Archer, M., Joyce, J. J., Levitt, K. N., and Windley, P. J., editors, *TPHOLs*, pages 59–62. IEEE Computer Society.

Ceriotti, M., Mottola, L., Picco, G. P., Murphy, A. L., Guna, S., Corra, M., Pozzi, M., Zonta, D., and Zanon, P. (2009). Monitoring heritage buildings with wireless sensor networks: The torre aquila deployment. In *Proceedings of the 2009 International Conference on Information Processing in Sensor Networks*, IPSN '09, pages 277–288, Washington, DC, USA. IEEE Computer Society.

Chakeres, I. D. and Belding-Royer, E. M. (2004). Aodv routing protocol implementation design. In *Proceedings of the 24th International Conference on Distributed Computing*

Systems Workshops - W7: EC (ICDCSW'04) - Volume 7, ICDCSW '04, pages 698–703, Washington, DC, USA. IEEE Computer Society.

Challen, G. and Welsh, M. (2010). Volcano monitoring: Addressing data quality through iterative deployment. In Guara, E., Girod, L., Brusey, J., Allen, M., and Challen, G., editors, *Wireless Sensor Networks: Deployments and Design Frameworks*. Springer.

Chatzigiannakis, I., Mylonas, G., and Nikoletseas, S. (2007). 50 ways to build your application: A survey of middleware and systems for wireless sensor networks. In *Emerging Technologies and Factory Automation, 2007. ETFA. IEEE Conference on*, pages 466–473.

Choi, Y.-S., Jeon, Y.-J., and Park, S.-H. (2010). A study on sensor nodes attestation protocol in a wireless sensor network. In *Proceedings of the 12th international conference on Advanced communication technology*, pages 574–579, Piscataway, NJ, USA. IEEE Press.

Chong, C.-Y. and Kumar, S. P. (2003). Sensor networks: evolution, opportunities, and challenges. In *Proceedings of the IEEE*, pages 1247–1256.

Clarke, E. M. (2008). 25 years of model checking. chapter The Birth of Model Checking, pages 1–26. Springer-Verlag, Berlin, Heidelberg.

Clarke, E. M. and Klieber, W. (2012). Model checking and the state explosion problem. 7682:1–30.

Clarke, E. M. and Wing, J. M. (1996). Formal methods: state of the art and future directions. *ACM Comput. Surv.*, 28(4):626–643.

Costa, P., Coulson, G., Gold, R., Lad, M., Mascolo, C., Mottola, L., Picco, G. P., Sivaharan, T., Weerasinghe, N., and Zachariadis, S. (2007). The RUNES middleware for networked embedded systems and its application in a disaster management scenario. In *Fifth Annual IEEE International Conference on Pervasive Computing and Communications (PerCom 2007), 19-23 March 2007, White Plains, New York, USA*, pages 69–78.

Costa, P., Coulson, G., Mascolo, C., Picco, G. P., and Zachariadis, S. (2005). The runes middleware: A reconfigurable component-based approach to networked embedded systems. In *Proceedings of the 16th Annual IEEE International Symposium on Personal Indoor and Mobile Radio Communications (PIMRC'05)*, Berlin (Germany).

Craigen, D., Gerhart, S., and Ralston, T. (1993). An international survey of industrial applications of formal methods. In Bowen, J. and Nicholls, J., editors, *Z User Workshop, London 1992*, Workshops in Computing, pages 1–5. Springer London.

Crow, J. et al. (1995). *NASA Formal Methods Specification and Verification Guidebook for Software and Computer Systems, Volume I: Planning and Technology Insertion.*

NASA Office of Safety and Mission Assurance, Washington, DC. Available at http://eis.jpl.nasa.gov/quality/Formal_Methods/.

Damchoom, K., Butler, M. J., and Abrial, J.-R. (2008). Modelling and proof of a tree-structured file system in event-b and rodin. In *ICFEM*, pages 25–44.

DeRoever, W. and Engelhardt, K. (1999). *Data Refinement: Model-Oriented Proof Methods and Their Comparison*. Cambridge University Press, New York, NY, USA.

Didioui, A., Bernier, C., Morche, D., and Sentieys, O. (2013). Harvwsnet:a co-simulation framework for energy harvesting wireless sensor networks. In *International Conference on Computing, Networking and Communications (ICNC)*, pages 808–812, San Diego, United States.

Dijkstra, E. W. (1969). Structured programming. In Naur, P. and Randell, B., editors, *Software Engineering: Report of a Conference Sponsored by the NATO Science Committee, Garmisch, Germany, 7-11 Oct. 1968, Brussels, Scientific Affairs Division, NATO*.

Doddapaneni, K., Ever, E., Gemikonakli, O., Malavolta, I., Mostarda, L., and Muccini, H. (2012). A model-driven engineering framework for architecting and analysing wireless sensor networks. In *Proceedings of the Third International Workshop on Software Engineering for Sensor Network Applications*, SESENA '12, pages 1–7, Piscataway, NJ, USA. IEEE Press.

Dong, J. S., Sun, J., Sun, J., Taguchi, K., and Zhang, X. (2008). Specifying and verifying sensor networks: An experiment of formal methods. In *Proceedings of the 10th International Conference on Formal Methods and Software Engineering*, ICFEM '08, pages 318–337, Berlin, Heidelberg. Springer-Verlag.

Dunkels, A., Gronvall, B., and Voigt, T. (2004). Contiki - a lightweight and flexible operating system for tiny networked sensors. In *Proceedings of the 29th Annual IEEE International Conference on Local Computer Networks*, LCN '04, pages 455–462, Washington, DC, USA. IEEE Computer Society.

Edmunds, A. (2013). Developments in code generation tools for event-b. In *Proceedings of the 4th Rodin User and Developer Workshop*, TUCS Lecture Notes. TUCS.

Edmunds, A. (2014). Code generation - tool developments. In *Proceedings of the 5th Rodin User and Developer Workshop*.

Fall, K. (2011). The *ns* manual formerly ns notes and documentation [online]. http://www.isi.edu/nsnam/ns/doc/ns_doc.pdf.

Farooq, M. O. and Kunz, T. (2011). Operating systems for wireless sensor networks: A survey. *Sensors*, 11(6):5900–5930.

Fehnker, A. and Gao, P. (2006). Formal verification and simulation for performance analysis for probabilistic broadcast protocols. In Kunz, T. and Ravi, S. S., editors, *Ad-Hoc, Mobile, and Wireless Networks, 5th International Conference, ADHOC-NOW 2006, Ottawa, Canada, August 17-19, 2006, Proceedings*, volume 4104 of *Lecture Notes in Computer Science*, pages 128–141. Springer.

Firesmith, D. (2007). Common requirements problems, their negative consequences, and the industry best practices to help solve them. *Journal of Object Technology*, 6(1):17–33. (column).

Fitzgerald, J., Larsen, P. G., Pierce, K. G., and Verhoef, M. (2013). A formal approach to collaborative modelling and co-simulation for embedded systems. *Mathematical Structures in Computer Science*, 23(04):726 – 750.

Fraboulet, A., Chelius, G., and Fleury, E. (2007). Worldsens: Development and prototyping tools for application specific wireless sensors networks. In *Proceedings of the 6th International Conference on Information Processing in Sensor Networks*, IPSN '07, pages 176–185, New York, NY, USA. ACM.

Gawanmeh, A. (2011). Embedding and verification of zigbee protocol stack in event-b. *Procedia Computer Science*, 5(0):736 – 741. The 2nd International Conference on Ambient Systems, Networks and Technologies (ANT-2011) / The 8th International Conference on Mobile Web Information Systems (MobiWIS 2011).

Gerhart, S., Craigen, D., and Ralston, T. (1994). Experience with formal methods in critical systems. *IEEE Software*, 11(1):21–28.

Gibbs, W. W. (1994). Software's Chronic Crisis. *Scientific American*.

Goguen, J. A. and Tardo, J. J. (1979). An introduction to obj: A language for writing and testing formal algebraic program specifications. pages 170–189.

Guttag, J. V. and Horning, J. J. (1993). *Larch: Languages and Tools for Formal Specification*. Springer-Verlag New York, Inc., New York, NY, USA.

Hall, A. (1990). Seven myths of formal methods. *IEEE Softw.*, 7(5):11–19.

Hall, A. (1999). Using formal methods to develop an atc information system. In Hinchey, M. and Bowen, J., editors, *Industrial-Strength Formal Methods in Practice*, Formal Approaches to Computing and Information Technology (FACIT), pages 207–229. Springer London.

Heimlich, O., Sailer, R., and Budzisz, L. (2010). Nmlab: A co-simulation framework for matlab and ns-2. *SIMUL*, 0:152–157.

Henricksen, K. and Robinson, R. (2006). A survey of middleware for sensor networks: State-of-the-art and future directions. In *Proceedings of the International Workshop*

on Middleware for Sensor Networks, MidSens '06, pages 60–65, New York, NY, USA. ACM.

Hill, J., Szewczyk, R., Woo, A., Hollar, S., Culler, D., and Pister, K. (2000). System architecture directions for networked sensors. *SIGPLAN Not.*, 35(11):93–104.

Hinchey, M. G. and Bowen, J. P. (1995). *Applications of formal methods*. Prentice-Hall international series in computer science. Prentice Hall, London, New York (N. Y.), Toronto (Ont.).

Hoang, T. S., Fürst, A., and Abrial, J.-R. (2011). Event-B patterns and their tool support. *Software & Systems Modeling*, pages 210–219.

Hoare, C. A. R. (1969). An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580.

Hoare, C. A. R. (1978). Communicating sequential processes. *Commun. ACM*, 21(8):666–677.

Holzmann, G. (2003). *Spin Model Checker, the: Primer and Reference Manual*. Addison-Wesley Professional, first edition.

Houston, I. and King, S. (1991). Cics project report experiences and results from the use of z in ibm. In Prehn, S. and Toetenel, W., editors, *VDM'91 Formal Software Development Methods*, volume 551 of *Lecture Notes in Computer Science*, pages 588–596. Springer Berlin Heidelberg.

Howard, Y., Gruner, S., Gravell, A. M., Ferreira, C., and Augusto, J. C. (2011). Model-based trace-checking. *CoRR*, abs/1111.2825.

Huth, M. and Ryan, M. (2004). Verification by model checking. In *Logic in Compute Science: Modelling and Reasoning about Systems*, pages 172–187. Cambridge University Press, New York, 2nd editio edition.

Imran, M., Said, A. M., and Hasbullah, H. (2010). A survey of simulators, emulators and testbeds for wireless sensor networks. In *2010 International Symposium of Information Technology (ITSim)*, pages 897–902.

Intana, A., Poppleton, M., and Merrett, G. V. (2013a). Adding value to WSN simulation through formal modelling and analysis. In Julien, C. and Wehrle, K., editors, *4th International Workshop on Software Engineering for Sensor Network Applications, SESENA 2013, San Francisco, CA, USA, May 21, 2013*, pages 24–29. IEEE.

Intana, A., Poppleton, M., and Merrett, G. V. (2013b). Proof-based formal methods for wsn development with simulation approach. In *Proceedings of the 4th Rodin User and Developer Workshop*, TUCS Lecture Notes. TUCS.

Intana, A., Poppleton, M., and Merrett, G. V. (2015). A model-based trace testing approach for validation of formal co-simulation models. In *Symposium On Theory of Modeling and Simulation*, TMS '15, Alexandria, VA, USA.

Intana, A., Poppleton, M. R., and Merrett, G. V. (2014). A formal co-simulation approach for wireless sensor network development. *ECEASST*, 70.

Ishikawa, K. (1976). *Guide to Quality Control, Asian Productivity Organization.* UNIPUB.

Jackson, M. A. (1983). *System Development.* Prentice Hall.

Jastram, M., Hallerstede, S., Leuschel, M., and Jr, A. G. R. (2010). An approach of requirements tracing in formal refinement. In *VSTTE*, volume 6217 of *Lecture Notes in Computer Science*, pages 97–111. Springer.

Jevtić, M., Zogović, N., and Dimić, G. (2009). Evaluation of wireless sensor network simulators. In *17th Telecommunication forum TELEFOR*, pages 1303–1306.

Jones, C. B. (1990). *Systematic Software Development Using VDM (2Nd Ed.).* Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

Kandt, R. K. (2003). Software requirements engineering: Practices and techniques. Technical report, Jet Propulsion Laboratory, California Institute of Technology.

Katelman, M., Meseguer, J., and Hou, J. C. (2008). Redesign of the LMST wireless sensor protocol through formal modeling and statistical model checking. In Barthe, G. and de Boer, F. S., editors, *Formal Methods for Open Object-Based Distributed Systems, 10th IFIP WG 6.1 International Conference, FMOODS 2008, Oslo, Norway, June 4-6, 2008, Proceedings*, volume 5051 of *Lecture Notes in Computer Science*, pages 150–169. Springer.

Kaur, A., Gulati, S., and Singh, S. (2012). A comparative study of two formal specification languages: Z-notation & b-method. In *Proceedings of the Second International Conference on Computational Science, Engineering and Information Technology*, CCSEIT '12, pages 524–531, New York, NY, USA. ACM.

Kiess, W. and Mauve, M. (2007). A survey on real-world implementations of mobile ad-hoc networks. *Ad Hoc Netw.*, 5(3):324–339.

Killian, C., Anderson, J. W., Jhala, R., and Vahdat, A. (2007). Life, death, and the critical transition: finding liveness bugs in systems code. In *Proceedings of the 4th USENIX conference on Networked systems design and implementation*, NSDI'07, pages 18–18, Berkeley, CA, USA. USENIX Association.

Knight, J., DeJong, C., Gibble, M., and Nakano, L. (1997). Why are formal methods not used more widely? In *Fourth NASA Formal Methods Workshop*. Citeseer.

Köpke, A., Swigulski, M., Wessel, K., Willkomm, D., Haneveld, P. T. K., Parker, T. E. V., Visser, O. W., Lichte, H. S., and Valentin, S. (2008). Simulating wireless and mobile networks in omnet++ the mixim vision. In *Proceedings of the 1st international conference on Simulation tools and techniques for communications, networks and systems & workshops*, Simutools '08, pages 71:1–71:8, ICST, Brussels, Belgium, Belgium. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

Korkalainen, M., Sallinen, M., Kärkkäinen, N., and Tukeva, P. (2009). Survey of wireless sensor networks simulation tools for demanding applications. In *Proceedings of the 2009 Fifth International Conference on Networking and Services*, ICNS '09, pages 102–106, Washington, DC, USA. IEEE Computer Society.

Kurki-Suonio, R. and Järvinen, H.-M. (1988). Action System Approach to the Specification and Design of Distributed Systems. In *IWSSD '89 Proceedings of the 5th international workshop on Software specification and design*, pages 34–40, New York, USA.

Lamport, L. (1994). The temporal logic of actions. *ACM Trans. Program. Lang. Syst.*, 16(3):872–923.

Lamport, L. (2000). *Chapter on TLA+ from Software specification methods : An overview using a case study.* Formal approaches to computing and information technology : FACIT. Springer, London.

Lamport, L. (2002). *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Langendoen, K., Baggio, A., and Visser, O. (2006). Murphy loves potatoes: experiences from a pilot sensor network deployment in precision agriculture. In *Proceedings of the 20th international conference on Parallel and distributed processing*, IPDPS'06, pages 174–174, Washington, DC, USA. IEEE Computer Society.

Lano, K. and Haughton, H. (1995). Formal development in b abstract machine notation. *Information and Software Technology*, 37(5-6):303 – 316.

Leuschel, M. and Butler, M. (2008). ProB: an automated analysis toolset for the B method. *Journal on Software Tools for Technology Transfer*, 10(2):185–203.

Levis, P., Lee, N., Welsh, M., and Culler, D. (2003). Tossim: Accurate and scalable simulation of entire tinyos applications. In *Proceedings of the 1st International Conference on Embedded Networked Sensor Systems*, SenSys '03, pages 126–137, New York, NY, USA. ACM.

Liu, X., Chen, Z., Yang, H., Zedan, H., and Chu, W. C. (1997a). A design framework for system re-engineering. *2013 20th Asia-Pacific Software Engineering Conference (APSEC)*, 0:342.

Liu, X., Yang, H., and Zedan, H. (1997b). Formal methods for the re-engineering of computing systems: A comparison. In *Proceedings of the 21st International Computer Software and Applications Conference*, COMPSAC '97, pages 409–, Washington, DC, USA. IEEE Computer Society.

Madden, S. R., Franklin, M. J., Hellerstein, J. M., and Hong, W. (2005). Tinydb: An acquisitional query processing system for sensor networks. *ACM Trans. Database Syst.*, 30(1):122–173.

Malik, Q. A., Laibinis, L., Truscan, D., and Lilius, J. (2010). Requirement-driven scenario-based testing using formal stepwise development. *International Journal On Advances in Software*, 3(1 2):147160.

Manna, Z. and Pnueli, A. (1992). *The Temporal Logic of Reactive and Concurrent Systems*. Springer-Verlag New York, Inc., New York, NY, USA.

Matousek, P., Rysavy, O., de Silva, G., and Danko, M. (2010). Combination of simulation and formal methods to analyse network survivability. In Perrone, L. F., Stea, G., Liu, J., Uhrmacher, A. M., and Villén-Altamirano, M., editors, *3rd International Conference on Simulation Tools and Techniques, SIMUTools '10, Malaga, Spain - March 16 - 18, 2010*, page 57. ICST/ACM.

Méry, D. and Singh, N. K. (2011a). Analysis of dsr protocol in event-b. In Dfago, X., Petit, F., and Villain, V., editors, *SSS*, volume 6976 of *Lecture Notes in Computer Science*, pages 401–415. Springer.

Méry, D. and Singh, N. K. (2011b). Automatic code generation from event-b models. In *Proceedings of the Second Symposium on Information and Communication Technology*, SoICT '11, pages 179–188, New York, NY, USA. ACM.

Mottola, L. and Picco, G. P. (2011). Programming wireless sensor networks: Fundamental concepts and state of the art. *ACM Comput. Surv.*, 43(3):19:1–19:51.

Mottola, L. and Picco, G. P. (2012). Middleware for wireless sensor networks: An outlook. *Journal of Internet Services and Applications*, 3(1):31–39.

Müller, A. (2009). VDM - The Vienna Development Method. Bachelor thesis in "formal methods in software engineering", Research Institute for Symbolic Computation (RISC), Johannes Kepler University Linz, Austria.

Muñoz, C. (1999). PBS: Support for the B-method in PVS. Technical Report SRI-CSL-99-01, SRI International.

Nair, S. and Cardell-Oliver, R. (2004). Formal specification and analysis of performance variation in sensor network diffusion protocols. In *Proceedings of the 7th ACM international symposium on Modeling, analysis and simulation of wireless and mobile systems*, MSWiM '04, pages 170–173, New York, NY, USA. ACM.

Nakajima, S. (2009). Abstraction Aided Model-Checking for Validation of Event-B Descriptions (preliminary). In *IM_FMT 2009*, 16 February.

Naur, P. and Randell, B., editors (1969). *Software Engineering: Report of a Conference Sponsored by the NATO Science Committee, Garmisch, Germany, 7-11 Oct. 1968, Brussels, Scientific Affairs Division, NATO.*

Niazi, M. and Hussain, A. (2011). A novel agent-based simulation framework for sensing in complex adaptive environments. *Sensors Journal, IEEE*, 11(2):404–412.

Ölveczky, P. C. and Thorvaldsen, S. (2007). Formal modeling and analysis of the ogdc wireless sensor network algorithm in real-time maude. In *Proceedings of the 9th IFIP WG 6.1 international conference on Formal methods for open object-based distributed systems*, FMOODS'07, pages 122–140, Berlin, Heidelberg. Springer-Verlag.

OMG (2012). Omg systems modeling language (omg sysml). version 1.3. [online]. http://www.omg.org/spec/SysML/1.3/.

OMNeT++ (2014). Omnet++ api reference 4.6. http://www.omnetpp.org/doc/omnetpp/api/.

Owre, S., Rushby, J. M., and Shankar, N. (1992). Pvs: A prototype verification system. In *Proceedings of the 11th International Conference on Automated Deduction: Automated Deduction*, CADE-11, pages 748–752, London, UK, UK. Springer-Verlag.

Perkins, C. E. and Royer, E. M. (1999). Ad-hoc on-demand distance vector routing. In *Proceedings of the Second IEEE Workshop on Mobile Computer Systems and Applications*, WMCSA '99, pages 90–100, Washington, DC, USA. IEEE Computer Society.

Picco, G. P. (2010). Software engineering and wireless sensor networks: happy marriage or consensual divorce? In *Proceedings of the FSE/SDP workshop on Future of software engineering research*, FoSER '10, pages 283–286, New York, NY, USA. ACM.

Pierce, K. G., Gamble, C. J., Ni, Y., and Broenink, J. F. (2012). Collaborative modelling and co-simulation with destecs: A pilot study. In *3rd IEEE track on Collaborative Modelling and Simulation, in WETICE 2012*, pages 280 – 285. IEEE-CS.

Raghavendra, C. S., Sivalingam, K. M., and Znati, T., editors (2004). *Wireless Sensor Networks*. Kluwer Academic Publishers, Norwell, MA, USA.

Ramanathan, N., Kohler, E., and Estrin, D. (2005). Towards a debugging system for sensor networks. *Int. J. Netw. Manag.*, 15(4):223–234.

Raychaudhuri, D., Seskar, I., Ott, M., Ganu, S., Ramachandran, K., Kremo, H., Siracusa, R. J., Liu, H., and Singh, M. (2005). Overview of the ORBIT radio grid testbed for evaluation of next-generation wireless network protocols. In *IEEE Wireless Communications and Networking Conference, WCNC 2005, March 13-17, 2005, New Orleans, Louisiana, USA*, pages 1664–1669. IEEE.

Reddy, A. M. V., Kumar, A. P., Janakiram, D., and Kumar, G. A. (2009). Wireless sensor network operating systems: a survey. *Int. J. Sen. Netw.*, 5(4):236–255.

Reed, D. P. (1980a). Rfc 761, transmission control protocol. https://tools.ietf.org/html/rfc761.

Reed, D. P. (1980b). Rfc 768, user datagram protocol. https://tools.ietf.org/html/rfc768.

Reisig, W. (1985). *Petri Nets: An Introduction*. Springer-Verlag New York, Inc., New York, NY, USA.

Rozier, K. Y. (2011). Linear Temporal Logic Symbolic Model Checking. *Computer Science Review*, 5(2):163–203.

Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., and Lorensen, W. (1991). *Object-oriented Modeling and Design*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

Rushby, J. (1997). Specification, proof checking, and model checking for protocols and distributed systems with PVS. Tutorial presented at FORTE X/PSTV XVII '97: Formal Description Techniques and Protocol Specification, Testing and Verification. Available, with specification files, at http://www.csl.sri.com/forte97.html.

Saghar, K., Henderson, W., Kendall, D., and Bouridane, A. (2010). Formal modelling of a robust wireless sensor network routing protocol. In *AHS*, pages 281–288.

Salehi Fathabadi, A. (2012). *An approach to atomicity decomposition in the Event-B formal method*. PhD thesis, University of Southampton, Electronics and Computer Science.

Samper, L., Maraninchi, F., Mounier, L., and Mandel, L. (2006). GLONEMO: Global and accurate formal models for the analysis of ad-hoc sensor networks. In *Proceedings of the First International Conference on Integrated Internet Ad hoc and Sensor Networks (InterSense'06)*, Nice, France.

Savicks, V., Butler, M. J., and Colley, J. (2014). Co-simulating event-b and continuous models via FMI. In *Proceedings of the 2014 Summer Simulation Multiconference, SummerSim 2014, Monterey, CA, USA, July 6-10, 2014*, page 37. SCS/ ACM.

Schneider, S. (2002). *The B-Method: An Introduction*. Palgrave, New York, USA.

Schulte, W. (2000). Why doesn't anyone use formal methods? In Grieskamp, W., Santen, T., and Stoddart, B., editors, *IFM*, volume 1945 of *Lecture Notes in Computer Science*, pages 297–298. Springer.

Schuts, M., Zhu, F., Heidarian, F., and Vaandrager, F. W. (2009). Modelling clock synchronization in the chess gmac WSN protocol. In Andova, S., McIver, A., D'Argenio, P. R., Cuijpers, P. J. L., Markovski, J., Morgan, C., and Núñez, M., editors, *Proceedings First Workshop on Quantitative Formal Methods: Theory and Applications, QFM 2009, Eindhoven, The Netherlands, 3rd November 2009.*, volume 13 of *EPTCS*, pages 41–54.

Sharma, O., Lewis, J., Miller, A., Dearle, A., Balasubramaniam, D., Morrison, R., and Sventek, J. (2009). Towards verifying correctness for wireless sensor network applications using insense and spin. In *In Proceedings of the 16th International SPIN Workshop on Model Checking Software*, volume 5578 of *SPIN '09)*, pages 223–240. Springer-Verlag.

Shimizu, R., Tei, K., Fukazawa, Y., and Honiden, S. (2012). Case studies on the development of wireless sensor network applications using multiple abstraction levels. In *Proceedings of the Third International Workshop on Software Engineering for Sensor Network Applications*, SESENA '12, pages 22–28, Piscataway, NJ, USA. IEEE Press.

Silva, R., Pascal, C., Hoang, T. S., and Butler, M. (2011). Decomposition tool for Event-B. *SPE*, 41(2):199–208.

Sohraby, K., Minoli, D., and Znati, T. (2007). *Wireless Sensor Networks: Technology, Protocols, and Applications*. John Wiley & Sons, Inc., USA.

Sommerville, I. (2011). *Software Engineering*. International Computer Science Series. Pearson.

Spivey, J. M. (1989). *The Z notation: a reference manual*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

Stankovic, J. A. (2004). Research challenges for wireless sensor networks. *SIGBED Rev.*, 1(2):9–12.

Stankovic, J. A. (2006). Wireless sensor networks [online]. https://www.cs.virginia.edu/~stankovic/psfiles/wsn.pdf.

Sundani, H., Li, H., Devabhaktuni, V. K., Alam, M., and Bhattacharya, P. (2011). Wireless sensor network simulators a survey and comparisons. *International Journal Of Computer Networks (IJCN)*, 2(5):249–265.

Titzer, B. L., Lee, D. K., and Palsberg, J. (2005a). Avrora: Scalable sensor network simulation with precise timing. In *Proceedings of the 4th International Symposium on*

Information Processing in Sensor Networks, IPSN '05, Piscataway, NJ, USA. IEEE Press.

Titzer, B. L., Lee, D. K., and Palsberg, J. (2005b). Avrora: Scalable sensor network simulation with precise timing. In *Proceedings of the 4th International Symposium on Information Processing in Sensor Networks*, IPSN '05, Piscataway, NJ, USA. IEEE Press.

Titzer, B. L. and Palsberg, J. (2005). Nonintrusive precision instrumentation of microcontroller software. *SIGPLAN Not.*, 40(7):59–68.

Tolle, G., Polastre, J., Szewczyk, R., Culler, D., Turner, N., Tu, K., Burgess, S., Dawson, T., Buonadonna, P., Gay, D., and Hong, W. (2005). A macroscope in the redwoods. In *Proceedings of the 3rd international conference on Embedded networked sensor systems*, SenSys '05, pages 51–63, New York, NY, USA. ACM.

Turon, M. (2005). Mote-view: A sensor network monitoring and management tool. In *Proceedings of the 2Nd IEEE Workshop on Embedded Networked Sensors*, EmNets '05, pages 11–17, Washington, DC, USA. IEEE Computer Society.

Utting, M. and Legeard, B. (2007). *Practical Model-Based Testing: A Tools Approach.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Valmari, A. (1998). The state explosion problem. In *Lectures on Petri Nets I: Basic Models, Advances in Petri Nets, the volumes are based on the Advanced Course on Petri Nets*, pages 429–528, London, UK, UK. Springer-Verlag.

Varga, A. (2011). Omnet++ user manual version 4.2.2 [online]. http://www.omnetpp.org/doc/omnetpp/Manual.pdf.

Vicente-Charlesworth, L. and Galms, S. (2011). On the development of a sensor network-based system for wildfire prevention. In Luo, Y., editor, *Cooperative Design, Visualization, and Engineering*, volume 6874 of *Lecture Notes in Computer Science*, pages 53–60. Springer Berlin Heidelberg.

Wang, A., Basu, P., Loo, B. T., and Sokolsky, O. (2009). Declarative network verification. In *Proceedings of the 11th International Symposium on Practical Aspects of Declarative Languages*, PADL '09, pages 61–75, Berlin, Heidelberg. Springer-Verlag.

Wang, B. and Baras, J. S. (2012). Integrated modeling and simulation framework for wireless sensor networks. In *Proceedings of the 2012 IEEE 21st International Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises*, WETICE '12, pages 268–273, Washington, DC, USA. IEEE Computer Society.

Wang, B. and Baras, J. S. (2013). Hybridsim: A modeling and co-simulation toolchain for cyber-physical systems. *Distributed Simulation and Real Time Applications, IEEE/ACM International Symposium on*, 0:33–40.

Washizaki, H., Koike, T., Namiki, R., and Tanabe, H. (2012). Reusability metrics for program source code written in c language and their evaluation. In Dieste, O., Jedlitschka, A., and Juzgado, N. J., editors, *PROFES*, volume 7343 of *Lecture Notes in Computer Science*, pages 89–103. Springer.

Werner-Allen, G., Lorincz, K., Johnson, J., Lees, J., and Welsh, M. (2006). Fidelity and yield in a volcano monitoring sensor network. In *Proceedings of the 7th symposium on Operating systems design and implementation*, OSDI '06, pages 381–396, Berkeley, CA, USA. USENIX Association.

Werner-Allen, G., Swieskowski, P., and Welsh, M. (2005). Motelab: A wireless sensor network testbed. In *Proceedings of the 4th International Symposium on Information Processing in Sensor Networks*, IPSN '05, Piscataway, NJ, USA. IEEE Press.

Wessel, K., Swigulski, M., K?pke, A., and Willkomm, D. (2009). Mixim: the physical layer an architecture overview. In Dalle, O., Wainer, G. A., Perrone, L. F., and Stea, G., editors, *SimuTools*, page 78. ICST/ACM.

Wilensky, U. (1999). NetLogo: Center for connected learning and computer-based modeling, Northwestern University. Evanston, IL. http://ccl.northwestern.edu/netlogo/.

Woo, A., Tong, T., and Culler, D. (2003). Taming the underlying challenges of reliable multihop routing in sensor networks. In *Proceedings of the 1st international conference on Embedded networked sensor systems*, SenSys '03, pages 14–27, NY, USA. ACM.

Xia, F., Kong, X., and Xu, Z. (2011). Cyber-physical control over wireless sensor and actuator networks with packet loss. In Mazumder, S. K., editor, *Wireless Networking Based Control*, pages 85–102. Springer New York.

Xia, F., Tian, Y.-C., Li, Y., and Sun, Y. (2008). Wireless sensor/actuator network design for mobile control applications. *CoRR*, abs/0806.1569.

Xian, X., Shi, W., and Huang, H. (2008). Comparison of omnet++ and other simulator for wsn simulation. In *ICIEA 2008: 3rd IEEE Conference on Industrial Electronics and Applications*, pages 1439–1443.

Yao, Y. and Gehrke, J. (2002). The cougar approach to in-network query processing in sensor networks. *SIGMOD Rec.*, 31(3):9–18.

Ye, W., Heidemann, J., and Estrin, D. (2002). An energy-efficient mac protocol for wireless sensor networks. In *Proceedings of the IEEE Infocom*, pages 1567–1576, New York, NY, USA. USC/Information Sciences Institute, IEEE.

Yeganefard, S. and Butler, M. (2010). Evaluation of a Guideline by Formal Modelling of Cruise Control System in Event-B. In *Proceedings of the Second NASA Formal*

Methods Symposium (NFM 2010), pages 182–191, 13-14 April 2010, Washington DC, USA.

Yick, J., Mukherjee, B., and Ghosal, D. (2008). Wireless sensor network survey. *Comput. Netw.*, 52(12):2292–2330.