A Model-based Trace Testing Approach for Validation of Formal Co-simulation Models

Adisak Intana, Michael R. Poppleton, and Geoff V. Merrett
Electronics and Computer Science
University of Southampton, Southampton, SO17 1BJ, UK
{ai1n10,mrp,gvm}@ecs.soton.ac.uk

ABSTRACT

This paper presents a model-based trace testing (MBTT) approach to strengthen verification and validation techniques for formal co-simulation based wireless sensor network development (*FoCoSim-WSN*). This framework enables the functionality and protocol algorithms to be encoded in the controller model in the formal Event-B language. Use of proof tools can guarantee safety properties of this formal model. Also, network reliability and performance analysis is performed by MiXiM simulation including e.g. the network load distribution and the network latency. However, this framework lacks focus in validation coverage since test scenarios for the controller model are generated randomly from the simulation environment. Consequently, the MBTT technique is applied to validate the formal Event-B controller in co-models. This technique enables us to create test scenarios 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 meeting our test requirements. The result shows how failing test scenarios in these runs ("killer traces") enable model debugging in terms of absent or erroneous constraints and events.

Author Keywords

co-simulation; formal methods; wireless sensor networks; model-based trace testing; validation coverage.

ACM Classification Keywords

D.2.4 SOFTWARE ENGINEERING: Software/Program Verification—*Formal methods,Validation*; D.2.5 SOFTWARE ENGINEERING: Testing and Debugging—*Tracing, Coverage Testing*; I.6.4 SIMULATION AND MODELING: Model Validation and Analysis

INTRODUCTION

A Wireless Sensor Network (WSN) is a distributed system of cooperating embedded devices called sensor nodes which are deployed to monitor and control physical real-world phenomena over a self-organised wireless network topology. Advances in low-power micro-embedded devices together with short-range wireless communication means that WSNs become increasingly adopted in a number of scientific fields e.g. environmental monitoring, traffic control, and habitat monitoring systems. Despite the powerful capability of this technology application, the successful development and deployment of WSNs is still a challenging task as the application is developed under the constraints of display-less, low-level specific platform and low power design [15, 3]. Furthermore, WSNs have encountered problems from real-world deployment concerning system failures that may be caused from node death or communication failures [3, 5]. Consequently, good software engineering (SE) methodologies and techniques are essential for WSN development.

Reliable Verification and Validation (V&V) is one of the most important techniques to provide a good SE practice for WSNs. The demand on V&V for WSN development is high since WSNs are potentially being deployed in safety-critical fields surrounded by harsh and remote environments. Therefore, several research studies have introduced various V&V techniques for WSN development, including formal methods [4, 18, 8] and experiment-based testing techniques such as simulation, emulation and testbeds [10, 7, 11]. Traditional pure formal development gives strong verification in which functional requirements and correctness properties - such as a loop-free property in the route tree - are defined formally as a safety invariant and proved that it is always satisfied [8]. On the other hand, experimental-based testing techniques provide confidence about reliability and performance analysis through long running simulation with a stochastic physical environment [10, 7, 11].

More recently, research studies have investigated a hybrid V&V approach between formal methods and simulation testing [13, 14, 9] to increase the strong V&V. In our preliminary work [9], we proposed a Formal Co-Simulation (*FoCoSim-WSN*) framework for WSNs providing an iterative interworking framework between proof-based formal verification, Event-B, and test-based simulation approaches, MiXiM. Formal methods provide the controller, a formal model of code in the real nodes, containing the protocol algorithms separately from the physical environment. An environment simulator provides stochastic sensed data and radio environment, allowing simulation scenarios to be defined as required. The result of this work confirms 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 con-

TMS/DEVS 2015 April 12-15, 2015, Alexandria, VA

troller model. Unexplored scenarios may still contain flaws e.g. incorrect constraints and events in the executable formal model. Consequently, the fine-gained testing approach supporting validation coverage by using a trace testing technique is introduced in this work.

The work we present in this paper is extended from our previous work [9]. The contribution of this work is to examine validation of formal controller models 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 that can influence the model debugging to reveal the absent or erroneous constraints and events.

The rest of this paper is organised as follow. Firstly, the next section provides an overview of background including the necessary approaches, methodologies and techniques we use in this work. Secondly, we describe a model-based trace testing (MBTT) approach for formal co-simulation. Then, the validation mechanism influenced by the MBTT process is expressed and demonstrated using a case study. This also describes "killer traces" for failure testing that enable the hidden incorrect constraints and events to be detected and debugged. Next, the results and related work are discussed. Finally, conclusions and areas of future work are summarised.

BACKGROUND

In this section, we introduce an overview of the languages, approaches and techniques we use in this work. This includes an overview of Event-B modelling and MiXiM simulation. Furthermore, our developed formal co-simulation (*FoCoSim-WSN*) framework is also expressed in this section.

Overview of Event-B Modelling

Event-B [1] is a proof-based formal method for specification and verification based on set theory and first order predicate logic. An Event-B model consists of two parts: *context* and *machine*. The context describing the static part contains *carrier sets*, *constants* and *axioms*. A machine representing the behavioural part includes the context by using the *sees* cause. The machine consists of three elements: *variables*, *invariants* and *events*. Dynamic behaviour in the model is defined by events updating variables. States are described by typed variables. Invariants that state the guaranteed properties of the model express the functional requirement and safety property. *Proof Obligations (POs)*, e.g. for maintenance of invariants, are generated from the model, and proved by RODIN theorem provers. A trace is defined by a sequence of executing events. Each event contains *guard(s), G(v)* and *action(s), A(v)*. The event guards express the necessary conditions which must be true to enable the event to successfully and usefully trigger, and actions describe the state transitions over the variables. An event with parameter t can be represented by the following form:

$$e \mathrel{\widehat{=}} \textbf{any } t \textbf{ when } G(t,v) \textbf{ then } A(v) \textbf{ end}.$$

Event-B Tool: RODIN [2] is an open tool platform based on Eclipse. This extensible tool was developed by the European Union ICT Project DEPLOY[1](2008-2012). RODIN includes editors, a proof obligation generator (PO-generator), graphical front ends, theorem provers and the *ProB*[2] animator and model checker.

Refinement is a method that allows software engineers to manage the complexity of the development by layering the abstraction of the models. A simple abstract view of essential requirements is implemented first. More requirement or design detail is added at each refinement step until implementation, data structure and algorithm are added to the concrete model in order to bring the model to become close to the real implementation. Refinement POs state that concrete refining events must correctly implement their counterpart abstract refined events.

Overview of MiXiM Simulation

MiXiM[3][11] provides a development framework for the simulation and performance analysis of wireless networks including WSNs. It provides generic and flexible component architecture for models based on a standard network simulation engine, OMNeT++[4]. Nodes in the simulation represent the wireless devices with their protocol stacks. Thus, the environment model is layered into the standard IP protocol stack. The data that has to be sent to a data sink is collected by *Application Layer (app)* before sending it down to lower layers. Next layer is the *Network Layer (net)* that manages the route tree used to decide the next hop for transmission towards the sink. The third layer is *MAC Layer (mac)* that manages power consumption by switching radio on/off to sending/receiving packets and provides an acknowledgement (ACK) mechanism for reliable transmission. The lowest layer, the *Physical Layer (phy)*, performs the radio propagation for packet sending and receiving.

MiXiM provides the base module (the general structure) that contains only communication interfaces named gates that create the connection between two adjacent layers. This base module can be extended to implement different protocol algorithms. The configuration file in MiXiM enables the specific protocol model for each layer to be identified and their parameter values to be assigned. This defines testing scenarios in order to analyse and evaluate non-function requirements of network e.g. the number of successful transmissions, network latency and network congestion.

FoCoSim-WSN framework

A Formal Co-Simulation (*FoCoSim-WSN*) [9] is a framework providing co-modelling and co-simulation for WSN development. This framework provides an iterative interworking scheme through multiple refinement levels between the formal node controller implemented by Event-B language and

[1] see DEPLOY - Industrial deployment of system engineering methods providing high dependability and productivity: FP7 Project 214158 - http://www.event-b.org

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

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

[4] see OMNeT++ -http://www.omnetpp.org/

physical environment and channel models on MiXiM simulation. This framework can be divided into three main components as shown in Figure 1.

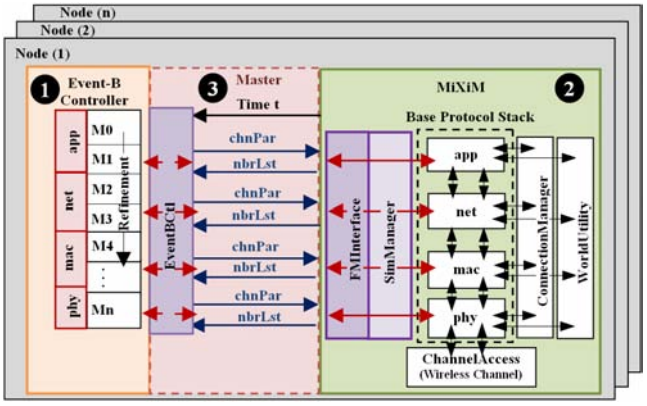

Figure 1. *FoCoSim-WSN* Co-simulation Framework

(1) The node software controller, representing the functional requirement, computation and algorithm for each protocol layer is developed and verified through refinement steps provided by Event-B and its RODIN toolkit. The safety property defined as a safety invariant guaranteeing the absence of certain classes of fault can be proved that it is always satisfied.

(2) MiXiM provides physical and communication environment for formal node controller models. The virtual node, node created at the simulation period by MiXiM, encodes the base protocol stack containing the gates for the communication between formal models and environment models e.g. physical and wireless channel models (*ChannelAccess*). The connection between each virtual node can be established by *ConnectionManager* and *WorldUtility*. Only nodes that are placed within the maximal distance interference can communicate to each other. This is used to identify who is the neighbour to receive a transmitted packet from the sender.

(3) A master, Groovy script - programmable API for ProB, co-simulates between formal node controller and simulation environment. Traces in Event-B controller model are implemented by an Event-B interface (*EventBCtl*) in the master. This class enables the instance of the Event-B model to be created and implemented as a thread for multithreading in the 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 MiXiM. These parameters are exchanged via a socket interface implemented in a master and MiXiM's front-end interface, named *FMInterface*. The packet parameter (*chnPar*) passed from *FMInterface* through back-end interface *SimManager* are transited down to/up from the wireless channel. The receiving neighbour list (*nbrLst*) is generated from the packet receiving information collected by *SimManager* and returned back to each node controller to perform receiving packet operation via *FMInterface* and master.

The co-modelling and implementation of this framework shows flexible mode of working. Two upper protocol layers, *SensorApp* for *application* and *MintRoute* for *network*, were modelled in Event-B and exercised with existing physical environment models provided by lower protocol models in MiXiM including MAC (implementing S-MAC), *Pathloss* and *Analogue* models.

MBTT APPROACH FOR FORMAL CO-SIMULATION

Figure 2 shows the Model-Based Trace Testing (MBTT) Approach for Formal Co-Simulation. This approach is a specification-based testing integrated from model-based testing [17] and trace checking [6] approaches. It aims to identify test traces for the controller model testing and validation.

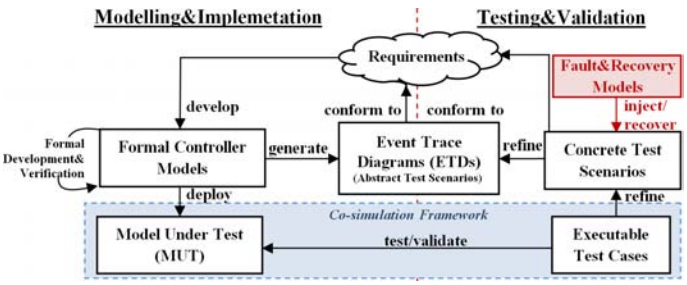

Figure 2. Model-based Trace Testing Approach for Formal Co-Simulation

In co-simulation modelling and implementation, 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, event trace diagrams (ETDs) are used to reduce the gap for translating a nonexecutable formal model to implementable concrete test scenarios. These diagrams express the sequence of events in the Event-B model representing the scenario of the controller behaviour. ETDs can be generated automatically from the master interface *EventBCtl*. Furthermore, the 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, this can derive the executable test case by adding the specific value of the parameter being able to find errors in a scenario execution.

MBTT IN ACTION

This section demonstrates the action taken in applying the MBTT approach proposed in the previous section to our case study's controller model. To accomplish this, we start with describing the controller modelling in Event-B in which two upper layer protocols, *SensorApp* and *MintRoute*, are layered through refinement steps. Then, we describe how we can analyse and formalise the event trace in the controller model

in order to create abstract and concrete test scenarios, respectively. Furthermore, test case production and the validation are demonstrated, before the experiment results are discussed. Finally, "killer traces" for detecting and debugging the hidden erroneous constraints and events are highlighted.

Event-B Controller Models

In our previous work [9], 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*[19], 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 *M1-M5* as shown in Figure 3.

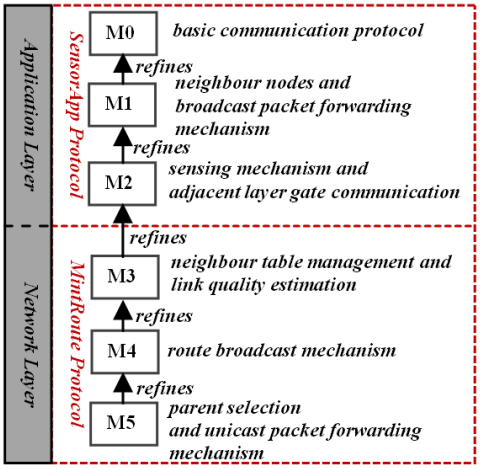

Figure 3. Event-B Controller Modelling Structure

SensorApp protocol modelling: an initial model *M0* develops *SensorApp* as a simple abstraction over the network protocol. Then, the first two refinement *M1* and *M2* fulfill 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.

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*. In this refinement model, one neighbour discovers another neighbour based on broadcasting a beacon packet. The receiving node will record the sender as a neighbour with the receiving information e.g. the number of receiving and lost packets. Furthermore, this refinement model implements *link quality estimation*. The estimation of the reception (inbound) link quality ratio is calculated periodically by observing the successful rate of receiving packets. As the link quality for both directions needs to be determined, the model *M4* introduces *route broadcast mechanism*. The reception link quality ratio needs to be attached in "*route update message*" and transmitted back to the neighbour. Finally, the model *M5* implements the *parent selection mechanism*, performing periodically to specify one of a 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). A link which has these two ratios less than the quality threshold is not considered. A neighbour with the smallest path cost is chosen as a parent.

Failure detection implementation: *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 over the time 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. Furthermore, this node is excluded during performing the parent selection mechanism. We implement dead neighbour detection mechanism and the *liveliness* maintenance mechanism in model *M4*.

Event-B Model Trace Analysis

Before creating test scenarios, we start analysing the event trace representing the controller behaviour. This can be expressed by the event trace diagram (ETD)[16]. Firstly, we extract the system scenarios represented in terms of the finite sequence of occurring events in the interface class *EventBCtl*. Then, ETDs are generated automatically from this interface class by an eclipse based UML creator and generator, ModelGoon[5]. 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 can be viewed as an abstract test scenario we use to create the concrete one described in the next section.

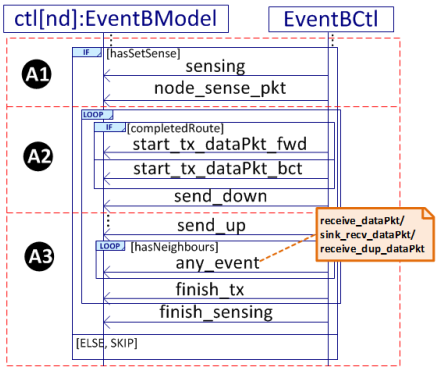

Figure 4. ETD describing the trace of the data sensing and transmission mechanisms

Figure 4 shows an example of generated ETDs describing the data sensing and transmission mechanisms for the *SensorApp* protocol. These mechanisms can be described as follows.

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

(A1) Only active sensor node (*hasSetSense==TRUE*) senses data periodically from the environment (shared event *sensing*). Then, a data packet containing sensed data is created (*node_sense_pkt*). (A2) if the node has a parent (*completedRoute==TRUE*), this parent will be specified as the next node to transmit such a created packet (*start_tx_dataPkt_fwd*). On the other hand, otherwise, it will assign the broadcast code (-1) for the next node destination to broadcast the data packet to all neighbours (*start_tx_dataPkt_bct*). Step (A3) demonstrates the packet transmission mechanism via the protocol stack. Each created/forwarded packet recorded in each sender/forwarder node's buffer is sent down (shared event *sent_down*) to the lower layer of the protocol stack and finally to the wireless channel, before this packet 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 (*receive_dataPkt* or *sink_recv_dataPkt*) or detect the incoming packet to be a duplicated packet (*receive_dup_dataPkt*).

Deriving Concrete Test Scenarios

As described in the earlier section, two types of testing are considered in MBTT approach for co-simulation, *functional* and *failure testing*. The former group of testing concerns on testing with a valid trace. Thus, we derive the abstract test scenarios that are five ETDs achieved from the previous section to manually identify the concrete test scenarios for the normal transmission and calculation (grouped into *TS-1*). These test scenarios fulfill information about the operation of Event-B controller and its environment implemented by MiXiM simulation. However, the automatic generation for this step is required for future work.

On the other hand, the latter testing involves forcing our co-simulation to behave consistently with fault scenarios. We apply the fault injection technique to achieve this testing goal. The fault model including the common failures in WSN systems after the deployment, node and link failures, together with the recovery mechanism are implemented in a simulation environment. 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 fault. To develop this, we extend each normal testing scenario described above by injecting the 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 5(a), some links surrounding transmission node (node 3) lose the connection.

TS-3: All link failures e.g. in Figure 5(b), all links surrounding transmitting node (node 3) lose the connection.

TS-4: Single dead node e.g. in Figure 5(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 5(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[6])

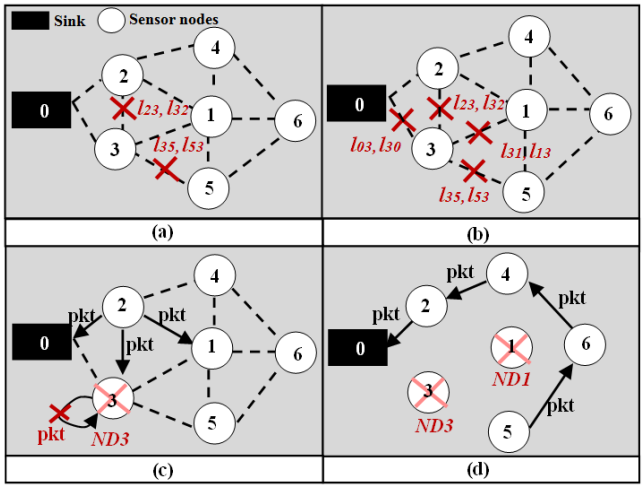

Figure 5. Different testing scenarios derived from ETDs

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. Figure 6 shows one of concrete test scenarios in scenarios *TS-2* describing partial link failure of node 3 as in Figure 5(a). This test scenario extends the abstract scenario described in Figure 4 with link failure injection. The scenario flow is described in Table 1.

As can be seen in Figure 6, we indicate some links surrounding the sender to be failed in order to drop the link quality ratio of failed link below the threshold. 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 2 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.

Designing and Creating Test Cases

The achieved testing scenarios provide us the guideline to design and create executable test cases that will be configured in

[6]the real deployment project used to derive our case study [8, 9]

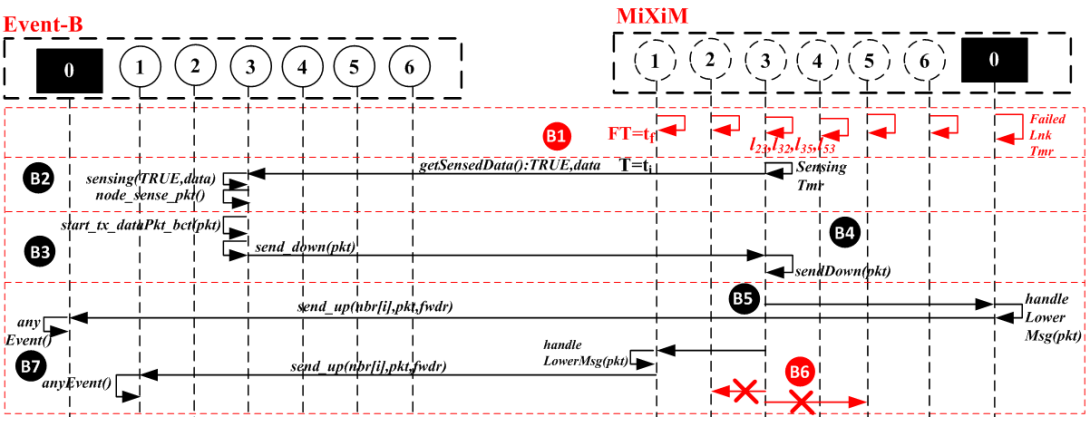

Figure 6. Testing scenario TS-2 initiating partial links (of node 3) to be failed (corresponding to Figure 5(a))

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 *node_sense_pkt* 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.

Table 1. Scenario flows of test scenario TS-2

MiXiM's configuration file of our co-simulation framework. We apply the MiXiM's testing methodology in which multiple instances of test scenarios are created to implement specific test cases configured in the configuration file. In this work, each instance of our test scenarios (*TS-1* to *-6*) is created based on 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 be calculated every 1, 2 or 3 packet transmission cycles).

As a result of this, *functional test cases* are derived from each instance of testing scenario *TS-1*, the normal transmission and calculation with the perfect network. For the failure and recovery test case purposes, instances of test scenarios related to the failure and recovery model (*TS-2 to -6*) are considered to derive test cases. Table 2 shows 8 test cases derived from one of test scenario *TS-2*'s instances for 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 illustrated 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.

TC#	Objective	Pass/Fail
TC-21	Detecting lost transmitted data packet from failed links.*	Pass
TC-22	Detecting lost transmitted beacon packet from failed links.**	Pass
TC-23	Checking cost estimation after some link fails.**	Pass
TC-24	Detecting lost transmitted route packets from failed links.**	Pass
TC-25	Checking *sentEst* recorded at cycle 3.**	Pass
TC-26	Checking *liveliness*.**	Pass
TC-27	Detecting dead neighbours.**	Pass
TC-28	Checking choose parent module.**	Fail

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

Table 2. Some test cases corresponding to test scenario TS-2

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 94% of the total number of test case results satified 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 occuring in the *MintRoute* algorithm as shown in Table 3.

TS#	# of Test Cases		
	Pass	Fail	Total
TS-1	6	0	6
TS-2	7	1	8
TS-3	3	1	4
TS-4	8	0	8
TS-5	4	0	4
TS-6	4	0	4

Table 3. Summary of results containing failed test cases

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-28* and *TC-32* regarding partial and all link failure scenarios respectively.

Considering test case *TC-28* in Figure 8(a), we injected links l_{23}, l_{32}, l_{35} and l_{53} corresponding to Figure 5(a) to be failed before each node broadcasts 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 *@g6* and *@g7* in Figure 7(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 7(b)).

@g6 receiveEst(x ↦ *nb*) < *th* @g7 sentEst(x ↦ *nb*) < *th*	@g6 receiveEst(x ↦ *nb*) < *th* ∨ sentEst(x ↦ *nb*) < *th*
(a)	**(b)**

Figure 7. (a) incorrect guards in *cal_PCost_le* and (b) the guard after correction

On the other hand, test case *TC-32* in Figure 8(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 5(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.

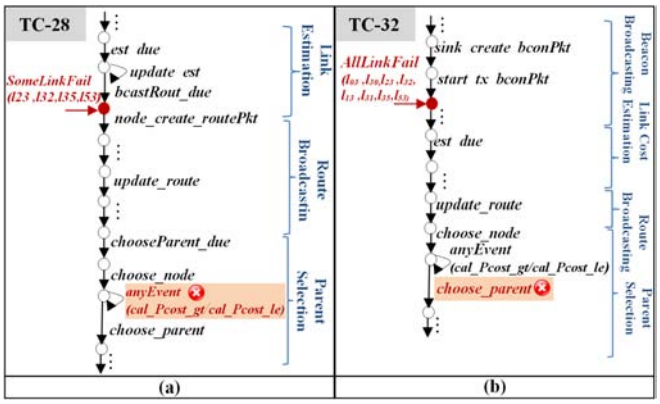

Figure 8. "Killer Traces"of test cases (a) *TC-28* and (b) *TC-32*, discovering the error along the trace execution

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 the "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.

RELATED WORK

Trace checking became integrated into model-based or model-driven software engineering paradigm to strengthen the relationship between modelling and testing. In [6], *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 satisfy the safety, liveness and deadlock properties 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. in [12] 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 the closest to our approach in which test scenarios are generated from Event-B models. However, ETDs are used in this work rather than process algebra to express test scenarios from Event-B models as their graphical representation is useful for understandability.

CONCLUSION AND FUTURE WORK

We have presented 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 scenario 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. Our results show that 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.

As the testing framework proposed in this work supports the semi-auto generation from the Event-B model controller to the 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 code generation from the node controller for the real node together with the extension of MBTT for validating generated SUT.

ACKNOWLEDGMENTS

The authors would like to thank the Royal Thai Government for funding a scholarship which made this research and paper possible.

REFERENCES

1. Abrial, J.-R. Formal methods : Theory becoming practice. *Journal of Universal Computer Science 13*, 5 (2007), 619–628.
2. Abrial, J.-R., Butler, M. J., Hallerstede, S., and Voisin, L. An open extensible tool environment for event-b. In *Proc. ICFEM 2006*, Springer Berlin Heidelberg (2006), 588–605.
3. Allen, M., Challen, G., and Brusey, J. Designing for deployment. In *Wireless Sensor Networks*, E. Guara, L. Girod, J. Brusey, M. Allen, and G. Challen, Eds. Springer, 2010.
4. Bernardeschi, C., and Masci, P. Early prototyping of wireless sensor network algorithms in pvs. In *Proc. SAFECOMP 2008*, Springer-Verlag (2008), 346–359.
5. Beutel, J., Roemer, K., Woehrle, M., and Ringwald, M. Deployment techniques for sensor networks. In *Sensor Networks - Where Theory Meets Practice*. Springer, 2010, 219–248.
6. Howard, Y., Gruner, S., Gravell, A. M., Ferreira, C., and Augusto, J. C. Model-based trace-checking. *Computing Research Repository abs/1111.2825* (2011).
7. Imran, M., Said, A. M., and Hasbullah, H. A survey of simulators, emulators and testbeds for wireless sensor networks. In *Proc. ITSim 2010* (2010), 897–902.
8. Intana, A., Poppleton, M. R., and Merrett, G. V. Adding value to wsn simulation through formal modelling and analysis. In *Proc. SESENA 2013*, IEEE (2013), 24–29.
9. Intana, A., Poppleton, M. R., and Merrett, G. V. A formal co-simulation approach for wireless sensor network development. *ECEASST 70* (2014).
10. Kiess, W., and Mauve, M. A survey on real-world implementations of mobile ad-hoc networks. *Ad Hoc Netw. 5*, 3 (Apr. 2007), 324–339.
11. Köpke, A., and et al. Simulating wireless and mobile networks in omnet++ the mixim vision. In *Proc. SIMUTools 2008*, ICST (2008), 71:1–71:8.
12. Malik, Q. A., Laibinis, L., Truscan, D., and Lilius, J. Requirement-driven scenario-based testing using formal stepwise development. *International Journal On Advances in Software 3*, 1 & 2 (2010), 147–160.
13. Matouek, P., Ryav, O., De, G. S., and Danko, M. Combination of simulation and formal methods to analyse network survivability. In *Proc. SIMUTools 2010*, ICST (2010), 6.
14. Niazi, M., and Hussain, A. A novel agent-based simulation framework for sensing in complex adaptive environments. *Sensors Journal, IEEE 11*, 2 (Feb 2011), 404–412.
15. Pietro, G. P. Software engineering and wireless sensor networks: Happy marriage or consensual divorce? In *Proc. FoSer 2010*, no. 4, ACM (2010), 283–286.
16. Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., and Lorensen, W. *Object-oriented Modeling and Design*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1991.
17. Utting, M., and Legeard, B. *Practical Model-Based Testing: A Tools Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2007.
18. Wang, A., Basu, P., Loo, B. T., and Sokolsky, O. Declarative network verification. In *Proc. PADL 2009*, Springer-Verlag (2009), 61–75.
19. Woo, A., Tong, T., and Culler, D. Taming the underlying challenges of reliable multihop routing in sensor networks. In *Proc. SenSys 2003*, ACM (2003), 14–27.