UNIVERSITY OF SOUTHAMPTON

FACULTY OF PHYSICAL AND APPLIED SCIENCES

Electronics and Computer Science

An Investigation into Event-B Methodologies and Timing Constraint Modelling

by

Gintautas Sulskus

Thesis for the degree of Doctor of Philosophy

September 2017

UNIVERSITY OF SOUTHAMPTON

ABSTRACT

FACULTY OF PHYSICAL AND APPLIED SCIENCES
Electronics and Computer Science

Doctor of Philosophy

AN INVESTIGATION INTO EVENT-B METHODOLOGIES AND TIMING CONSTRAINT MODELLING

by Gintautas Sulskus

In the domain of formal modelling and verification of real-time safety-critical systems, our focus is on complex – i.e. nested, interdependent and cyclic – timing constraints. We strengthen the pallet of modelling tools and techniques to describe and verify timing properties in such real-time systems. Our contribution builds on Event-B – a formal language for systems modelling, based on set theory and predicate logic. This language has the advantage of mechanised proof and tackles system complexity through a stepwise refinement.

The foundation of our scheme – a notion of a timing interval – is a higher-level abstraction in terms of state machine and formal timing interval specification. It can formally bind together several related timing requirements, expressed in delay, deadline and expiry concepts. To support the timing interval abstraction through the Event-B based refinement method, we present five compositional refinement transformations: *Sub-Interval, Alternative, Abort-to-Response, Single-to-Multi* and *Retry*. The timing interval and its refinement transformations use a template-based generative scheme for the transformation of timed models – specified with state machines and timing interval – to Event-B. We provide a workflow and a recommended convention for modelling and graphically representing a timing interval and its refinement transformations in state machine diagrams. The modelling of a timing interval and its refinement transformations process is automated with our tool – tiGen.

Finally, we validate our approach and the recommended development workflow in three case studies. The results show that our timing interval can be developed through multiple levels of refinement. The process of modelling and proving is mostly automated.

Contents

List of Figures

List of Tables

Declaration of Authorship

I, Gintautas Sulskus , declare that the thesis entitled *An Investigation into Event-B Methodologies and Timing Constraint Modelling* 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: [148] and [149]

Signed:.....

Date:.....

Acknowledgements

I would like to express my sincere gratitude to my supervisors Dr. Michael Poppleton and Dr. Abdolbaghi Rezazadeh for their continuous support throughout the four years of my PhD, their input during countless meetings, and their encouragement and guidance during difficult times. Special thanks to my internal examiner Prof. Michael Butler for providing valuable suggestions and corrections.

Thanks to all the colleagues of the ESS group for the friendly and uplifting atmosphere, motivating attitude and work environment that made the lab feel like a second home.

I would like to thank my family and friends for their understanding, support and encouragement throughout these four years of work.

Lastly, I would like to thank again Dr. Michael Poppleton for his remarkable patience and for going way beyond the call of duty in all the help he provided.

Chapter 1

Introduction

A modern pacemaker is a computer with specialised peripherals. Hardware of the Airbus A380 airliner is controlled solely by a computerised system that interprets a pilot's commands. Features such as autopilot, lane departure warning and collision detection systems are gradually making cars more autonomous. Moreover, communication between such systems is becoming more important, morphing into a bigger distributed network. As we become increasingly dependent on such systems, the consequences of their failures become more severe. We call such systems *safety-critical* [114].

Most safety-critical systems are *real-time* [63]. In real-time systems it is not enough to just have a correct reaction, the reaction must also happen within specified boundaries of time. Typically, these boundaries are not negotiable – they are determined by the environment they operate in. Therefore, the specifications of real-time systems include many time-related requirements [147].

Real-time safety-critical systems may have a number of properties that increase the complexity and reduce the confidence in the correctness thereof. Concurrent and communicating components tend to exhibit unpredictable interactions that may lead to incorrect behaviours [83]. The errors are often caused by the fact that the inherent non-determinism makes the behaviour complex and hard to predict. The number of ways the system can behave becomes very large, and thus it is hard to validate that it will operate as intended in a given situation. Time is one of the things that makes this process more complicated for real-time systems than others. The time-based scheduling of real-time systems adds extra details; thus the system is made even more complex [115].

Conventional systems engineering methods do not provide a sufficient level of confidence in real-time safety-critical system correctness [90]. The quantifications of such system reliability are intractable – they inevitably lead to a need for testing beyond what is practical [51]. The failure rate of software development projects in the last three decades

remained at around 70% [70]. A substantial body of information systems literature has proposed and tested a list of both technological and organisational factors that should assist organisations in successfully completing projects and delivering expected benefits [56]. However, given the persistence of such high failure rates, it appears that substantial accumulated knowledge has not made a difference in information systems practice. Such failed projects waste many billions of dollars of organisational resources annually [70, 108].

A number of disasters resulting from flaws in system design have led to significant financial loss and fatalities. Between 1985 and 1987 a Therac-25 medical electron accelerator caused a number of fatal accidents that are attributed to race conditions and arithmetic overflows in the software – this allowed the device to bypass safety checks and activate in an error setting [126]. At the time, the system had no hardware-based safety redundancy to add protection against software errors. The United States General Accounting Office issued a report [153] on a Patriot missile defence system failure in Saudi Arabia that failed to protect soldiers from Scud missiles. It was concluded that a bug in the system software caused a less accurate time calculation after a prolonged system operation. Over time, this bug eventually lead to a loss of accuracy. In 1996, an unmanned Ariane 5 rocket launched by the European Space Agency exploded forty seconds after lift-off. The rocket and its cargo were valued at $500 million. The disaster was caused by the poor reuse of the rocket code from Ariane 4 and a failure to apply software-related measures to handle unexpected conditions [72]. With time it became apparent that a more rigorous approach is needed to ensure that faults do not lead to disasters.

Formal techniques are becoming more important in addressing the aforementioned design issues [90, 116]. In essence, formal methods are the application of applied mathematics – in this case, formal logic – to the design and analysis of software-intensive systems [141]. As mathematics is a clear language, formal methods help to articulate requirements in a clear way. Expressed specification may then be verified and checked by proof and model checking. Different formal techniques suit different forms of mathematical analysis and verification. These techniques help to eliminate ambiguities and provide a high level of assurance so that all stakeholders can agree on a consistent definition.

It is expected that, like in other engineering disciplines, formal methods would contribute to the robustness and reliability in real-time safety-critical system design. Typically, a specific formal language is selected and used to express system requirements in a mathematical form so one can prove the models consistency. In general, the process of formalising real-time safety-critical systems is challenging and not trivial [113, 87]. The modeller must understand or work closely with an expert of a problem domain to produce a document specifying sophisticated requirements. In many cases, systems are concurrent and have interconnected modular architecture that introduces further complexity. Moreover, time requirement specification, verification and integration with other modelled system aspects is a non-trivial task and is development-wise costly [143].

Examples of proposed formal methods for real-time systems include, among others, timed transition systems/temporal logic [105, 75, 136, 95], timed automata [23], timed Petri-Nets [85] and proof-based verification methods such as Isabelle [10], Z [21], VDM [81], B method [54] and Event-B [19]. Regarding proof and verification, it is common practice to investigate hybrid solutions to get the best of both worlds and to overcome the existing limitations of specific methods. Such examples are timed, probabilistic and hybrid automata [22, 162, 24] or a combination of several verification approaches such as proof-based Event-B with automated STM solvers [65] and a ProB model-checker [125].

1.1 Research Questions

The key problem that this thesis addresses is that the modelling of timing requirements in Event-B is mostly based on patterns that lack a higher-level abstraction and expressibility. There is a need for flexible modelling techniques that would be applicable out of the box for a wide range of applications without requiring case-specific customisations. This section elaborates on the limitations we have observed with existing approaches. The observed points identify four key research areas.

RQI. Higher Level Abstraction. Modelling timing constraints in a methodical and reusable fashion has not been investigated much in the Event-B community. The formalism lacks native timing support [78] and research has been done to address this limitation [55, 109, 143]. A proposed timing constraint pattern [55] is tightly coupled with the model structure and is relatively low level. [143] classifies timing constraints into expiry, delay and deadline, similar to patterns in timed automata [156]. While the latter approach is methodical and reusable, we have observed that it can be further extended regarding modelling flexibility. Other formal methods that we have overviewed in chapter 2 lack a level of abstraction and do not provide support for refinement. Our particular interest is concurrent cyclic systems, where timing requirements overlap and interconnect. For this, we need to lift modelling to a higher abstraction level so that it would allow us to reason about them collectively. The collection can then be refined as a whole.

RQII. Refinement. Refinement plays a central role in Event-B formalism. It allows one to deal gradually with system requirement complexity by adding details in a stepwise manner. For systematic and modular reuse of refinement, refinement transformations must consist of well-defined syntactic constructs of the modelling language. Regarding this, [143] has advanced the most in developing timing refinement patterns for Event-B. However, the given patterns are quite limited in choice and flexibility. New transformations are needed to give a modeller freedom to manipulate and "sculpt" the requirements through refinement.

RQIII. A Method. Sarshogh [143] has proposed a textual notation to specify individual timing constraints. The notation needs to be extended to formally define aggregate requirements, such as combining delay and deadline together. A higher-level representation maps better to model-based software development techniques such as Unified Modelling Language (UML) state machines [17]. Moreover, a visual outline of the specified requirements can give a better understanding of timing requirement relations and refinement hierarchies. Visual representation provides a better abstraction and overview of the system. Graphical representation is widely used in timed automata [23] and in toolboxes based on it such as UPPAAL [122]. However, the timed automata method does not support refinement and state nesting. A tool called iUML supports refinement and translates UML state machine diagrams to Event-B. We are interested in methodical and validated ways to specify rich timing requirements and their refinements graphically.

RQIV. Generative Approach. [143] provides well-defined syntactic Event-B constructs that apply in a generative fashion. There is, however, a lack of investigation into if and how these constructs could be categorised and reused to produce new refinement patterns. We look into generic pattern-like Event-B constructs that can potentially provide core building blocks for future transformation and facilitate the generative approach and tool development.

1.2 Research Scope

Our research is limited to a horizontal refinement development phase and does not consider data refinement. In this phase, the model of a system is incrementally elaborated with given requirements. We do not consider decomposing the system into several communicating sub-systems, as this typically comes after the horizontal refinement.

The functional part of the system is not our primary focus. Therefore, the case studies that we have used to validate our approach have a limited level of detail regarding the functional aspect.

Since the process of Event-B to implementation code generation, such as ADA, Java or C [130, 73, 82, 140], is quite well investigated in the Event-B community, we do not cover this phase of system development.

1.3 Contribution

Our contribution is inspired by time modelling methodologies used in timed automata and the work that had been done already in Event-B. We analyse existing Event-B modelling patterns and approaches and propose a methodology of modelling timing

requirements for real-time systems. An overview of related work is covered in section 3.6, and the differences between that work and our contribution is discussed in detail in section 12.4.

We provide a scheme that enables a modeller to reason about the timing requirements at a higher level of abstraction. The scheme comprises methodology and mechanised modelling constructs for timing requirement specification, validation and verification in Event-B. We split the scheme into the following five key parts.

I. Timing Interval. The foundation of our scheme – a notion of a timing interval (RQI). It is a higher-level abstraction that can formally bind together several related timing requirements. The new approach uses an extended notation to specify timing requirements and builds on delay, deadline and expiry concepts proposed by [143]. This timing interval empowers the modeller to reason upon a collection of requirements as a unitary entity. The approach is based on a trigger-response pattern and adds a special kind of response event called abort. Design decisions make full state-space coverage model-checking feasible on smaller models. The approach supports multiple interdependent and cyclic timing constraints. We investigate the relation and consistency between multiple distinct timing requirements and their refinements. We consider them in terms of enabledness and deadlock freedom.

II. Refinement Transformations. We propose five refinement transformations for our timing-interval approach (RQII). Two of them are well known as the alternative and sub-interval refinement transformations. One pattern is transformative in the sense that it does not produce new timing intervals but rather modifies the already existing one. Two transformations are based on process transformation when a simple interval is transformed into a number of sub-processes. The transformations allow a modeller to start off with the atomic, abstracted timing requirement specification. The specification can then be gradually refined with more elaborate requirements while maintaining consistency across the refinements. The patterns are interoperable, and the refinement depth is not limited.

III. Template-Based Constructs. We take a generative design approach to facilitate the modelling, verification and automation of the system. The timing interval and its refinement transformations are defined as a set of generic Event-B specification templates (RQIV). Depending on the timing interval specification, relevant templates are instantiated, combined and injected into the Event-B model. A subset of the templates is shared among some of our refinement transformations. We have developed a tool, called tiGen, to provide the means for specifying timing intervals and their refinements, and then automate Event-B specification generation at any stage of modelling. The tool provides a visual outline of the timing interval refinement structure.

IV. Modelling Methodology. We provide a recommended convention to graphically represent a timing-interval and its refinement transformations in iUML state machine diagrams. Moreover, we formalise the synchronisation between timing interval and the iUML [11] state machine diagrams. The guidelines enable the modeller to reason in a visual model-based and methodical development style about the system's behaviour and timing requirements (RQIII).

We demonstrate our new approach capabilities in three case studies. They exploit new timing interval approach capabilities, such as event overloading, multiple trigger support and abort events (chapter 4), and formally integrate with the state machines. The pacemaker case study is heavy on cyclic and overlapping timing intervals and demonstrates how a single abstract interval can be refined into multiple concrete and concurrent timing intervals. The message passing protocol case study primarily aims to investigate multi-instance timing interval applicability and more complex refinement patterns. To further validate the timing interval approach, we have applied it to a case study of landing gears. The models were fully proven and partially model-checked, with no inconsistencies found. The majority of the development was automated with iUML and tiGen tools. The manual elements are specified in the case study chapters.

1.4 Outline

This thesis is structured into twelve chapters. The first two chapters represent the background of this work:

- Chapter 2 overviews a number of existing methods for system verification and validation and gives detail on the foundation of formal methods and timed automata.
- Chapter 3 focuses on Event-B formalism. We give more detail on Event-B language and describe the tools used in this work. Further, we overview the research done on timing aspect in Event-B and elaborate on Sarshogh's work since we reuse his concept of timing properties. We then discuss the work done on timing refinement in Event-B. Finally, we discuss the verification and validation of temporal properties in Event-B.

We present our contribution in the following chapters:

- Chapter 4 introduces our timing interval approach and its notation. We provide the motivation for our approach and highlight the advantages thereof compared to Sarshogh's work. We then, in a simple example, demonstrate the approach and explain its semantics. We describe the automation process and provide generative templates for it.

- Chapter 5 presents five refinement transformations for the timing interval. The transformations are explained through simple examples. Generative aspects and templates are given. We then discuss the approach regarding enabledness and deadlock freedom.
- In chapter 6 we overview the plug-in that we developed for the Rodin platform to generate timing interval and its refinements. A brief manual and plug-in architecture are provided.
- Chapter 7 provides modelling conventions for iUML and timing-interval approach integration.

The contribution is validated in the following chapters:

- In Chapter 9 we give detail on the pacemaker case study modelled with the timing interval approach.
- Chapter 10 covers the message passing case study that evaluates the practicality of the timing interval and the multi-instance timing-interval feature in particular.
- In Chapter 11 we conduct a third case study – the landing gear system – to verify our claims about the capabilities of the timing interval approach.
- Finally, chapter 12 concludes.

Chapter 2

Research Landscape

This chapter gives an overview of modelling and verification approaches and points out the advantages and shortcomings thereof. We cover model-checking and deductive verification methods and some of the existing combinations thereof in more detail. Further, we concentrate on foundation topics that underlie the formal verification methods that we use in our research. Finally, we give details on timed automata and some of its modelling patterns. In further sections we contrast these with the patterns we use in Event-B.

2.1 Conventional Techniques

The problem with the programming languages is that they fail to provide an easier representation of business or requirement rules because by design they describe algorithmic and lower-level technical computer commands [151]. The need for further abstraction arose for targeting domain problems but not execution commands. The result was an introduction of software models – an abstract representation of challenges at hand. Model-driven development is centred around the usage of a model's simplified representation of a certain system. Models are further used to automate the process of software development, partially or fully [151]. The model-driven software development, which is primarily focused on the vertical separation of concerns, is aimed at reducing the gap between problem and software implementation domains through the use of models that describe complex systems at different abstraction levels and from a variety of perspectives.

2.1.1 Model-Based Development

We briefly overview some of the industry accepted standards and tools that are well-aligned with the model based software development vision.

UML [17] is a de facto standard in object-oriented software development. Real-time embedded systems often use extensions of a subset thereof, called profiles. Such profile examples are SysML [100], Modelling and Analysis of Real-time Embedded Systems (MARTE) [12] and Executable UML (xUML) [129]. SysML is a standard general-purpose language for systems engineering. It allows capturing of the interactions with the physical world in a mathematical model, and the verification of properties on it. MARTE is intended to add modelling capabilities to verify real-time properties such as timeliness and schedulability [64]. Executable UML is the evolution of the Shlaer-Mellor method [146, 145] for object-oriented structured analysis. The latter can be viewed as a further evolution of classical structured analysis for real-time systems [157]. xUML "combines" a subset of the UML graphical notation with executable semantics and timing rules. Models of xUML are typically executed for simulation and are precise enough to be translated to the implementation code. Architecture Analysis & Design Language (AADL) is a de facto standard in the domain of avionics and automotive software systems. The use of AADL enables various types of analyses that link to dependability and safety aspects.

A wide range of tools support the mentioned languages. Papyrus [14] is a tool for UML/SysML modelling. It is part of a greater suite for systems engineering – PolarSys (formerly TOPCASED) [15] – designed for real-time embedded system projects. Abstract Solutions [8] is a tool for xUML that was used to support a model-driven architecture approach to the mission software for the F-16 fighter. AADL is supported by open-source and commercial tools such as Stood [16].

While these techniques contribute to safety, they are all focused on a particular phase of the software development life-cycle [100]. Modelling language specialities, such as the physical modelling capabilities of SysML, functional modelling of UML, runtime architectural modelling of AADL (and its analysis capability) and MARTE can be combined [64, 128, 110] to improve requirements specification and traceability. Unless used with formal verification tools, model-based development methods are prone to conventional verification limitations.

2.1.2 Verification

One of the key issues in software development is ensuring that the delivered product meets its specification. With models being expressed in UML, the application of verification and validation is complicated. Firstly, concerning verification, a UML model is typically not the input language of a verification tool. Secondly, with regard to validation, a UML model is also not directly executable.

Verification by testing is impossible for modelled systems that have to operate in what has been called the ultra-dependable range. Yet in practice, testing remains the principal

approach to verifying systems [155]. This method requires fewer mathematical skills and is cheaper and faster to implement than other methods [134]. Tests can only show the presence of errors in a software system – not the absence thereof – and cannot show that a system is free of design flaws, nor can they test software for correctness. That is, they can show the presence of bugs, but not that software is free of design errors. Due to these reasons, it is difficult to apply tests for concurrent, distributed and real-time systems. Testing is used in later stages of system development; hence, addressing found bugs is more expensive in terms of labour and cost.

Using a computer simulation against a program can replace prototyping, which as a result reduces the risk of design errors. However, it is neither possible nor practical to test all system traces [86]. An example language for simulation is Modelica [74], which is supported by the open source tool OpenModelica [13].

Static program analysis is the automatic compile-time determination of run-time properties in a program [155]. Static analysis builds an abstract representation of a program's behaviour and examines its states. The analyser maintains extra information about a checked program, which is an approximation of the real one. Because of these approximations, static analysis tools may miss weaknesses (false negatives) or report correct code as having a weakness (false positives). Static timing-analysis techniques are also used on time-critical applications to estimate the worst-case execution times [158]. However, these techniques too suffer from approximation. An example static analysis tool is SLAM [31].

Runtime Verification (RV) is a dynamic analysis method aimed at checking whether a run of the system under scrutiny satisfies a given correctness property [79]. The advantage of RV is that it can be applied to an already existing software. Most systems can be runtime verified, but the verification only ensures that some important properties hold for the fraction of the system that happens to be checked during verification.

The main technical problems in runtime verification are automatically adding runtime verification code to an existing system and minimising the overheads of that code. A popular approach for the first problem is to use aspect-oriented programming techniques that allow such code to be written independently of the original system and through compiler techniques automatically injected into the code [96]. An example of the aspect-oriented extension is AspectJ [111] for JAVA. For the second problem, some work has employed static analysis to eliminate any runtime checks that can easily be identified as unnecessary [96].

2.2 Formal Methods

Formal methods are mathematical techniques, that are often supported by tools, for developing software and hardware systems [161]. Mathematical rigour enables users to analyse and verify these models at any part of the program lifecycle: requirements engineering, specification, architecture, design, implementation, testing, maintenance, and evolution.

The development of a formal specification provides insight and understanding of software requirements and software design [90]. This reduces requirement errors and omissions. It provides a basis for an elegant software design. Using formal methods, it may be possible to prove specification consistency and completeness. It may also be possible to prove that an implementation conforms to its specification.

Some specifications come with a framework of ready-to-use tools that facilitate the system specification and verification process. Some frameworks support system validation via means of animation, where the mathematical model can be "run" and manipulated by the modeller [125].

In the following sections, we discuss two general techniques of formal methods: model checking and deductive verification.

2.2.1 Model Checking

Model checking is a method where one takes a model and a specification of a system expressed as a temporal logic formula, and algorithmically checks whether the system satisfies the property [61, 60]. Typically, the property is a formula in some temporal logic that returns a Boolean result indicating whether or not the model satisfies the specification. These formulas are checked against the model state space – a tree of all possible execution paths and states. The advantage of model checking is that it is fully automatable and, in cases of inconsistency, provides a counter example for easy traceability [139]. While model checking is commonly applied to finite-state systems, it is also possible to apply it to infinite-state systems using, for example, symmetry properties, abstractions and small-model properties [96]. Example systems are: ProB [125] model checker for Classical B; SPIN [99] translates Promela description language into non-deterministic automata for model checking; UPPAAL [122] that is based on timed automata theory; AlPiNA [44] which uses Petri nets to describe system behaviour.

Some model checkers use stochastic methods for calculating the likelihood of the occurrence of certain events during the execution of a system. In common with conventional model checking, stochastic model checking involves reachability analysis of the underlying transition system, but it must also entail the calculation of actual likelihoods through appropriate numerical or analytical methods [117]. Examples of statistical

model checking are UPPAAL-SMC [45], discrete-time Markov chains (DTMC) [58] and continuous-time Markov chains (CTMCs) [30].

State-explosion remains the main limitation of using model checking for large systems. Although research has advanced to a level where some infinite-state systems have been model checked, it is still often the case that the state space of a system that is being modelled is too large, and no known reduction in the state space removes this problem. Due to this, model checking is mainly used as a debugging mechanism in the early design stages, with methods such as bounded model checking gaining popularity [96].

One way to avoid state-space explosion in a concurrent program is to decompose it into processes and model check each component separately, then piece the results back together to conclude that the original program is correct. However, model checking techniques are not easily amenable to a composition and fail to preserve critical properties. While much work has been done on compositional model checking [96], the application of such methods in real-world-scenario software systems is still impractical.

2.2.2 Deductive Verification

Deductive verification techniques employ symbolic representation and thus do not suffer directly from model checking state explosion. Avoiding the size limitation makes deductive methods much more suitable for designs of a larger scale [139].

However, deductive methods have fundamental weaknesses of their own. The main disadvantage is the fact that they are not fully automatic and require the modeller to have a good mathematical knowledge to complete the proofs [142]. Another disadvantage is that deductive methods do not produce a counterexample.

In comparison to model checking, deductive reasoning is more general, but it is only partially automatable and requires a higher user-competence in mathematics.

Some examples of formalisms employing deductive verification are VDM [80], Z[69], Classical B [18] and its evolution Event-B [19].

2.2.3 Combination of Verification Methods

Since each of the mentioned verification methods has its strengths and weaknesses, it is tempting to combine them to leverage the advantages of each.

For example, SLAM static analysis toolkit uses a combination of predicate abstraction, model checking, symbolic reasoning and iterative refinement to statically check temporal safety properties of programs [32].

While model checking only works well when the entire system can be examined, a runtime verification requires a single run of the system. This difference makes the combination of the two techniques attractive. [119] utilised the combination of model checking and runtime verification to identify security vulnerabilities in web applications. [52] utilises runtime analysis to perform an any-time variant of static analysis. [39] employed static program analysis to simplify runtime verification checks.

Hansen et. al. [91] used a combination of Event-B and ProB to model a case study of a landing gear. While Event-B is good at infinite-state-space systems and invariant safety property verification, it struggles with temporal property verification and deadlock safety property does not scale well [163]. Therefore, a ProB model-checker was used by [91] to verify temporal properties which are not expressible in Event-B. Moreover, Hansen et. al. used a ProB animator to validate their model in the form of test cases. [36] combined deductive verification with a software-bounded model checking (SBMC). Model checker helped in finding early and precise feedback about insufficient or wrong proof-based specifications.

2.2.4 Practicality

Industrial grade formalisms have already been applied in real-time system development in industry. VDM was used to model the software requirements and functional design of an air traffic control [89] for London Air Traffic Control Centre. OpenComRTOS is one of the few real-time operating systems (RTOS) for embedded systems that was developed using formal modelling techniques [154]. Authors of [112] successfully verified a general purpose operating system called kernel *seL4* using Isabelle/HOL prover. Since 2011, engineers at Amazon have been using TLA+ [121] to help solve difficult design problems in critical systems [135]. A driver-less metro line 14 in Paris, a shuttle for Paris-Roissy and an automatic train protection for the French railway company SNCT are examples of a B-Method application [29, 37]. Formalism was used to verify requirement specifications and translation to ADA implementation language [9]. A fully automated Storm Surge Barrier Control System [42] to protect the Netherlands against floods was partially specified and validated with Spin tool set [98] and Z formalism [160]. Functional aspects (no timing) of the control software for radiation therapy machines were formally verified by [104]. Deploy [6] and Advance [7] projects bring together academia and industry to improve and promote formal verification techniques – the real-world applications include the formal verification of railway signalling and smart grid systems.

2.2.5 Dijkstra's Guarded Command Language

Event-B, the formal language used in this work, is heavily influenced by action systems. Since the roots of Action Systems partially lie in Dijkstra's Guarded Command Language

(GCL), it is important to introduce the key principles of it.

GCL is defined by Dijkstra but presented as a rather general (theoretical) concept. Programs in GCL act on variables, are sequential and intended to be terminating [67].

The most important element of the language is the guarded command. In GCL, commands are guarded by predicates that must be *true* prior to execution. Pre-condition blocks help to prove if the program meets the specification.

Having predicate guard blocks enables alternative and repetitive constructs that allow non-deterministic program components for which activities, and possibly even the final state, are not necessarily determined by the initial state.

In order to prove the total correctness of a program, Dijkstra introduced the *weakest precondition* (2.1), also known as the *predicate transformer*, that is part of *Weakest Precondition Calculus*. Weakest precondition defines the weakest possible pre-condition *pre*; that must be met so that after S statement's execution state space would always be guaranteed to satisfy the post-condition *post*.

$$pre \Rightarrow wp(S, post) \tag{2.1}$$

Dijkstra defines *wp* by saying: "The condition that characterizes the set of all initial states, such that activation will certainly result in a properly terminating happening leaving the system in a final state satisfying a given post-condition, is called 'the weakest pre-condition corresponding to that post-condition"' [66].

For example, the weakest pre-condition for the statement $y := y^2$ with a post-condition $\{y < 10\}$ would be $\{y < 4\}$, given that $y \in \mathbb{N}$:

$$\{y < 3\} \Rightarrow wp(y := y^2, \{y < 10\}) \tag{2.2}$$

[67] has expressed GCL formally:

$$
\begin{aligned}
\langle \textit{guarded command} \rangle &::= \langle \textit{guard} \rangle \rightarrow \langle \textit{guarded list} \rangle \\
\langle \textit{guard} \rangle &::= \langle \textit{boolean expression} \rangle \\
\langle \textit{guarded list} \rangle &::= \langle \textit{statement} \rangle \, \{; \, \langle \textit{statement} \rangle\} \\
\langle \textit{guarded command set} \rangle &::= \langle \textit{guarded command} \rangle \, \{[] \, \langle \textit{guarded command} \rangle\} \\
\langle \textit{alternative construct} \rangle &::= \textit{if} \langle \textit{guarded command set} \rangle \textit{fi} \\
\langle \textit{repetitive construct} \rangle &::= \textit{do} \langle \textit{guarded command set} \rangle \textit{od} \\
\langle \textit{statement} \rangle &::= \langle \textit{alternative construct} \rangle \mid \langle \textit{repetitive construct} \rangle \\
&\quad \mid \text{“}\textit{otherstatements}\text{”}
\end{aligned}
$$

Braces {...} read as “followed by zero or more instances of the enclosed”.

This expression may be seen hierarchically. The top element is a *guarded command set* that consists of *guarded commands*. Separator [] mutually separates *guarded commands*. The order of the list is semantically irrelevant. A *guarded command* is a relation of one to many, where a Boolean expression *guard* must be true for a *guarded list* to be executed. A *Guarded list*, as the name states, is a list of statements to be executed. A semicolon denotes that execution order is important. The statement is either *alternate construct*, *repetitive construct* or *“other statement”*. The latter covers assignment statements and procedure calls.

In the alternative construct case, if in the initial state none of the guards is true, the program will abort. Otherwise, an arbitrary guarded list with a true guard will be selected for execution. Then the repetitive guard case, all of the guarded lists that have their guards true will be executed. When the repetitive construct has terminated properly, we know that all of its guards are false.

An example of a repetitive construct program could be a simple loop with two variables x and y and arbitrary initial values X and Y (2.3). Inside the construct, there are two guarded commands. The first command $x := x - y$ is enabled when condition $\{x > y\}$ is true. The second command $y := y - x$ is enabled when condition $\{y > x\}$ is true. The construct terminates when two variables, x and y, are equal. Hence none of the two guarded commands has guard predicate enabled.

$$
\begin{aligned}
& x := X; \;\; y := Y; \\
& \textit{do} \; x > y \rightarrow x := x - y \\
& [] \quad y > x \rightarrow y := y - x \\
& \textit{od}.
\end{aligned}
\tag{2.3}
$$

When expressed in more traditional clauses, the program would look like this (2.4):

$$
\begin{array}{l}
x := X; \quad y := Y; \\
\textbf{while } x \neq y \textbf{ do} \\
\qquad \textbf{if } x > y \textbf{ then } x := x - y \\
\qquad\qquad \textbf{else } y := y - x \\
\qquad \textbf{fi} \\
\textbf{od}.
\end{array}
\tag{2.4}
$$

A further two important elements remain unmentioned: *skip* and *abort*. The skip instruction defines "do nothing" and never affects state space; thus, precondition is identical to postcondition $pre = wp(skip, post)$ where $pre = post$. On the other hand, the *Abort* instruction has a completely arbitrary behaviour; hence, the post-condition is always $false - pre = (abort, false)$ [67].

2.2.6 Refinement Calculus

Today, approaches to formally engineered software systems [41] are predominately based either on the ideas of a design-by-contract [131] or on the refinement calculus methodology [133]. Methods based on design-by-contract, such as Java Modelling Language [123], use contracts to specify the interface between software modules in first-order logic. The client module must satisfy the conditions specified in the contract before calling a provider module; in return, the provider guarantees certain properties that will hold after the call. The implementation of the modules are then verified against those contracts. In refinement calculus approaches, such as B [18]) and Event-B [19], systems are modelled gradually, starting from the abstract model and then elaborating the model with more features via a series of property preserving steps. Eventually, the model is brought to the implementation version. Most of the modelling effort goes into proving that a model and its transformation are models of the same system. The contribution of this thesis builds on the Event-B formalism and the general principles of the refinement calculus.

The refinement calculus is a logical framework for reasoning about programs, introduced by Back and Wright [27]. The framework is concerned with two main questions: 1) whether a program is correct with respect to a given specification, and 2) how we can improve, or refine, a program while preserving its correctness.

The refinement calculus framework originated from the stepwise refinement method for program construction by Dijkstra [68] and Wirth [159]; and the transformational approach to programming [84, 46], and early work on program correctness and data refinement by [97]. The purpose of the refinement calculus is to provide a solid logical

foundation for these methods, based on the weakest precondition approach to the total correctness of programs proposed by Dijkstra [66].

In the refinement calculus, an agent is a process that can act and choose. It could be human, a computer or processes in a computer system. These agents are non-deterministic since they appear as a black box and may not be predicted with certainty.

It is assumed that the world is described as a state σ and is part of a state space Σ. The agent may change the state by applying function f to yield a new state $f.\sigma$. Such an action is denoted as $\langle f \rangle$.

The contract is what regulates the behaviour and interaction of the agents. The contract may enforce a specific order of actions that the agent must carry out [27]. A simple language for the contract may be denoted as such:

$$S ::= \langle f \rangle \mid S_1;\ S_n \mid S_1 \sqcup S_n \tag{2.5}$$

Where $\langle f \rangle$ reads as a state-changing action (e.g. variable value assignment): $S_1;\ S_n$ is a stipulated list of statements that agent needs to perform, while $S_1 \sqcup S_n$ is an alternative action where the agent must choose any of the given choices S_n.

Also, agents have requirements called *assertions*. These (2.6), that must be met in a given state. Otherwise, the contract will be breached. The assertion is expressed as $\{g\}$, where g is a condition on a state. On the other hand, *assumptions* (2.7) are agent expectations. If the assumption does not hold, the agent is released from the contract. Assumptions are expressed as $[g]$.

$$S ::= x := 1;\ \{x > 1\};\ X := x - 1 \tag{2.6}$$

$$S ::= x := 1;\ [x > 1];\ X := x - 1 \tag{2.7}$$

The refinement calculus specifies that the agent "can satisfy contract S to establish postcondition q in initial state σ if the agent can either achieve a final state that satisfies q without breaching the contract or is released from the contract by an assumption that is violated" [27]. Such a statement is denoted by 2.8:

$$\sigma \{|\ S\ |\} q \tag{2.8}$$

Let's say we have two contract statements S and S' for our agent. Then, if all conditions that may be established with S can also be established with S', we say that S is *refined* by S'. We denote this by $S \sqsubseteq S'$. A formal way of expressing it is as follows:

$$\sigma \{| S |\} q \Rightarrow \sigma \{| S' |\} q \text{ , for any } \sigma \text{ and } q \tag{2.9}$$

The refinements are transitive, meaning that if $S \sqsubseteq S'$ and $S' \sqsubseteq S''$ then $S \sqsubseteq S''$

This theory leads us to a central application – a stepwise refinement of programs. It is a top-down program specification principle. In simple terms, one has to start with a high-level system specification and then gradually refine it by a series of correctness-preserving transformations into an efficient and concrete executable program (2.10). An individual step refinement $S_i \sqsubseteq S_i + 1$ can be established by adding additional specification to a statement:

$$S_0 \sqsubseteq S_1 \sqsubseteq ... \sqsubseteq S_n \tag{2.10}$$

Based on transitivity, $S_0 \sqsubseteq S_n$.

The goal of such an approach is to be able to alter the system with the guarantee that each new refinement layer preserves the correctness of all preceding refinements up to the highest level (initial) specification.

2.2.7 Temporal Logic

Originally, temporal logic was developed to be used in philosophy and could be split into two major branches – modal and tense logic.

Modal logic became the custom amongst philosophers in the first half of the twentieth century to characterise different views about necessity and possibility [53].

Modal logic extends classical propositional and predicate logic; it uses modal words that express modalities to qualify a statement. Two basic modals, the necessity and possibility, are usually written as $\Box$ and $\Diamond$ respectively.

Tense logic is an extension of propositional logic. Classically, propositional formulas are interpreted as truth values (either true or false). Once we know the value, it is fixed. The basic idea of temporal logic is to make valuations time-dependant. To deal with this, tense logic covers two ideas.

The first idea is to relate propositional logic to time. Then propositional logic with time formally looks like 2.11, where T is a time flow, and ϕ is a set of propositional variables.

$$T \to (\phi \to \{1, 0\})) \tag{2.11}$$

The second idea is to introduce tense operators P and F. These letters are mnemonics for 'past' and 'future' respectively. $F\phi$ would read as "at some time in the future ϕ holds", and $P\phi$ as "at some time in the past ϕ holds".

In the second half of the twentieth century, [47] and [138] combined modal and tense logics and proposed temporal logic to be used in computer science as an appropriate formalism in the specification and verification of concurrent programs.

Computational Tree Logic (CTL) and Linear Temporal Logic (LTL) are two classes of modern temporal logics. The fundamental difference is that CTL views the computation of a system as a tree, whereas an LTL system has only one direction into the future.

Let Φ be a specification, then CTL syntax can be given in Backus-Naur form as [102]:

$$\begin{aligned} \Phi ::= &\bot \mid \top \mid p \mid (\neg\Phi) \mid (\Phi \wedge \Phi) \mid (\Phi \vee \Phi) \mid (\Phi \to \Phi) \mid \\ &AX\Phi \mid EX\Phi \mid AF\Phi \mid EF\Phi \mid AG\Phi \mid EG\Phi \mid A[\Phi U \Phi] \mid E[\Phi U \Phi] \end{aligned} \tag{2.12}$$

Notice that in CTL temporal connectives are pairs of symbols, where the preceding symbol is always a temporal quantifier A or E. A means along all paths (inevitably), E means along at least one path (possibly), X means next state, F means some future state, G means all future states (globally) and U means until.

LTL does not have branches and thus lacks temporal quantifiers A and E [102]. A temporal connective in LTL is a single symbol. Two additional are present: R – release and W – weak until:

$$\begin{aligned} \Phi ::= &\bot \mid \top \mid p \mid (\neg\Phi) \mid (\Phi \wedge \Phi) \mid (\Phi \vee \Phi) \mid (\Phi \to \Phi) \mid \\ &(X\Phi) \mid (F\Phi) \mid (G\Phi) \mid (\Phi U \Phi) \mid (\Phi W \Phi) \mid (\Phi R \Phi) \end{aligned} \tag{2.13}$$

Formula $(a\ W\ b)$ means that a remains $\top$ until b becomes $\top$ but does not require that b ever does become $\top$. $(a\ R\ b)$ means that that b is $\top$ until a becomes $\top$, or b is $\top$ forever.

2.2.8 Safety, Fairness and Liveness

In the verification of programs, fairness assumption and two properties are of primary importance – *safety properties* and *liveness properties*. To explain them, we express these properties with LTL.

Safety Safety properties usually state that something bad never happens [124]. Consider ω as a safety property, such as the absence of a deadlock or system required invariant. Then, a safety property must always hold:

$$G\omega \tag{2.14}$$

Fairness An action system that models a distributed system as a set of actions, each of which is either enabled or disabled [76]. A fairness assumption controls the selection of actions from this set for execution. Three fairness assumptions are defined. Let ϕ and ψ be propositional logic formulas. Think of ϕ as stating that "action A is enabled" and of ψ as defining "A action"[1].

- *Absolute* fairness constraint states that action ψ occurs infinitely often:

$$GF\psi \tag{2.15}$$

- *Strong* fairness constraint states that if action ψ is infinitely often enabled ($GF\phi$), then it has to occur infinitely often:

$$GF\phi \rightarrow GF\psi \tag{2.16}$$

- *Weak* fairness constraint: if action ψ is continuously enabled ($FG\phi$) then it has to occur infinitely often:

$$FG\phi \rightarrow GF\psi \tag{2.17}$$

If a property is *true* under the assumption of absolute or strong fairness, then it is also *true* under weak fairness. However, absolute fairness does not imply strong fairness, nor vice versa:

$$\text{Absolute Fairness} \Rightarrow \text{Weak Fairness}, \text{Strong Fairness} \Rightarrow \text{Weak Fairness} \tag{2.18}$$

An example fairness requirement is that given the time window, the heart always contracts, either intrinsically or paced:

$$GF(t_{lower} \leq t \leq t_{upper}) \rightarrow GF(H_{intrinsic} \vee H_{pacemaker}) \tag{2.19}$$

[1] Dr. Corina Cirstea, examples based on "Formal Design of Systems" lecture slides, 2011

Where H is the activity in the heart, occurring intrinsically or pacemaker-induced and t is the time of occurrence, that must be within predefined upper and lower time boundaries t_{lower} and t_{upper} respectively.

Liveness Liveness properties define the requirement for a process to make progress towards a specific goal [76]. If a process never gets to execute, it usually cannot reach its goal. Therefore, the achievement of the goal depends on the fairness assumptions of the system. Consider a requirement ξ and its fulfilment ϵ. Then, ξ will always eventually be fulfilled (ϵ):

$$G(\xi \rightarrow F\epsilon) \tag{2.20}$$

An example of a liveness property is the assumption (2.21) that the heart H will eventually contract when the contraction time t is within the allowed lower and upper time boundaries:

$$G((t_{lower} \leq t \leq t_{upper}) \rightarrow F(H_{intrinsic} \wedge H_{pacemaker})) \tag{2.21}$$

2.3 Timed Automata

Timed automaton is an extension of a finite automaton. It is essentially a directed graph with finite labelled edges and real-valued clocks [23]. It has been used for modelling and verifying systems which are triggered by events and have timing constraints between events. Formally, a timed automaton is a tuple $A = (\Sigma, S, I, C, E)$ that consists of the following components [23]:

- Σ is a finite alphabet (or actions) of A.
- S is a finite set of the states of A.
- $I \subseteq S$ is a set of start states.
- C is a finite set of clocks of A.
- $E \subseteq S \times S \times \Sigma \times 2^C \times \Phi(C)$ is a set of edges, called transitions of A, where
  - $\Phi(C)$ is the set of Boolean clock constraints involving clocks from C
  - 2^C is the power-set of C.
- An edge $s \xrightarrow{\sigma:a,R} s' \in E$ is a transition from state s to s'. Action a is enabled once guard σ holds. Upon transition execution, given clocks R are reset.

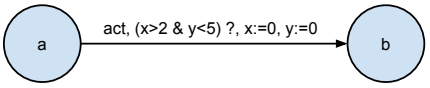

Figure 2.1: An example of timed automaton

Consider the example timed automaton in Figure 2.1. The automaton consists of states a and b and a transition act. Having a formal definition of timed automaton, we can specify our example as $A = (act, \{a, b\}, a, \{x, y\}, E)$, where E contains act specification. The transition is constrained by clocks x and y with the condition $x > 2 \wedge y < 5$. Upon transition execution, both clocks are set to 0. Formally the transition is specified as $a \overset{x>2 \wedge y<5 : act, \{x,y\}}{\longrightarrow} b \in E$

Timed automaton accepts *timed words* – infinite sequences in which a real-valued time of occurrence is associated with each action [23].

Formally, a *time sequence* $\tau = \tau_1, \tau_2 ...$ is an infinite sequence of $\mathbb{R}, \tau_1 > 0$ values, satisfying the following constraints [23]:

- *Monotonicity*: τ increases strictly monotonically – $\tau_i < \tau_{i+1}$, for all $i \geq 1$
- *Progress*: for every $t \in \mathbb{R}_+$, there is some $i \geq 1$ such that $\tau_i > t$.

A timed word over an alphabet Σ is a pair (σ, τ), where $\sigma = \sigma_1 \sigma_2 ...$ is an infinite word. A set of words over Σ form a *timed language* over Σ. If a timed word (σ, τ) is viewed as an input to an automaton, it presents the action σ_i occurrence at time τ_i.

To explain the timed language, consider the following example [23]. Let the alphabet be a, b. Define language L_1 to consist of all timed words (σ, τ) such that there is no b after time 5. Then, the language L_1 is given as:

$$L_1 = \{\sigma, \tau \mid \forall\, i \cdot ((\tau_i > 5) \rightarrow (\sigma_i = a))\} \tag{2.22}$$

2.3.1 Timed Automata Patterns

A timed automaton is essentially a flat finite-state machine equipped with clocks. Practical systems are typically complex; they therefore require additional language extensions and patterns to facilitate the modelling process and its interpretation. [156] presents a number of common timed patterns for timed automata, four of which are related to our research: *delay*, *deadline* and *time-out* patterns.

Parallel Composition The parallel composition pattern is one of the most useful patterns for timed automata [156]. The pattern defines multiple timed automata that run in parallel.

We extend timed automata tuple with a new element Inv. Inv is a function that associates automaton state to an invariant: $L : S \to \Theta(C)$. Where $\Theta(C)$ is the invariant for specified clocks.

Let $A_i = (\Sigma_i, S_i, I_i, C_i, Inv_i, E_i)$, where $i \in \{1, 2\}$ is two timed automata, where $C_1 \cap C_2 = \varnothing$. Parallel composition of A_1 and A_2 is written as $A_1 \| A_2$ and denoted graphically as shown in Figure 2.2.

$A_1 \| A_2$ timed automaton (Σ, S, I, C, E) has such properties: $S = S_1 \times S_2$; initial state is a pair of A_1 and A_2 initial states $I = \{(i, j) \mid i \in I_i \wedge j \in I_2\}$; $\Sigma = \Sigma_1 \cup \Sigma_2$; $C = C_1 \cup C_2$; invariant of any state in $(s_1, s_2) \in S$ is a conjunction of invariants of corresponding states in A_1 and A_2 – $\forall (s_1, s_2) \cdot (s_1, s_2) \in S \Rightarrow Inv((s_1, s_2)) = Inv_1(s_1) \wedge Inv_2(s_2)$.

A transition E can be either a local transition of either A_1 or A_2, or synchronisation transition on a common event of A_1 and A_2. E follows the following rules:

1. Local A_1 transitions are not affected by and do not affect A_2 timed automaton:

$$s_1 \xrightarrow{\sigma:a,R} s_1' \wedge a \notin \Sigma_2, \text{ then } s_1, s_2 \xrightarrow{\sigma:a,R} s_1', s_2 \text{ for all } s_2 \in S_2 \tag{2.23}$$

2. Analogous rule is applied for local A_2 transitions:

$$s_2 \xrightarrow{\sigma:a,R} s_2' \wedge a \notin \Sigma_1, \text{ then } s_1, s_2 \xrightarrow{\sigma:a,R} s_1, s_2' \text{ for all } s_1 \in S_1 \tag{2.24}$$

3. Common (synchronisation) transitions of A_1 and A_2 automata can be fired only when the constraints of both transitions are satisfied. After the execution, clocks related to both transitions are reset:

$$s_1 \xrightarrow{\sigma_1:a,R_1} s_1' \text{ and } s_2 \xrightarrow{\sigma_2:a,R_2} s_2', \text{ then } s_1, s_2 \xrightarrow{\sigma_1 \wedge \sigma_2:a,R_1 \cup R_2} s_1', s_2' \tag{2.25}$$

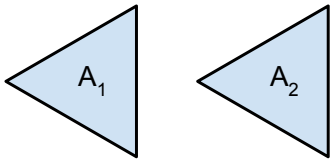

Figure 2.2: Parallel composition pattern

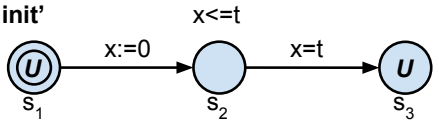

Figure 2.3: Delay pattern

Delay The delay pattern is used to delay the execution by exactly t time units and is notated as $delay(t)$, where $t \in \mathbb{R}_+$ [156].

An example delay pattern is a timed automaton $(\Sigma, S, I, C, Inv, E)$ (Figure 2.3) such that $S = s_1, s_2, s_3$; $I = s_1$; $\Sigma = \varnothing$; $C = \{x\}$ and $Inv(s_1) = urgent$ and $Inv(s_2) = x \leq t$ and $Inv(s_3) = urgent$. E contains two transitions $s_1 \overset{true:\tau,\{x\}}{\longrightarrow} s_2$ and $s_2 \overset{x=t:\tau,\varnothing}{\longrightarrow} s_3$. The state invariant *urgent* expresses the statement that time cannot pass while in such state. Event τ is an empty action.

The delay pattern in Figure 2.3 comes into effect in state s_2. The guard on transition $s_2 \overset{x=t:\tau,\varnothing}{\longrightarrow} s_3$ enforces the delay of t before event $s_2 \overset{x=t:\tau,\varnothing}{\longrightarrow} s_3$ can be executed. In this delay pattern version, the invariant on s_2 enforces the transition to happen after t runs out.

Deadline The deadline pattern is used to capture the requirement that some task must be finished by a certain time [156]. Given timed automaton A (Figure 2.4), if it is constrained to finish within t time units, all states in A are labelled with an invariant $x \leq t$ in which x is a new clock which is reset upon the control entering the automaton. Graphically (Figure 2.4) we represent all automaton A states as S with the deadline invariant above.

To express the deadline pattern formally, let $A = (\Sigma, S, I, C, Inv, E)$ be a timed automaton. Then the deadline pattern, written as $deadline(A, n)$ is a timed automaton $(\Sigma', S', I', C', Inv', E')$, where $S' = S \cup \{init'\}$; $I' = \{init'\}$; $\Sigma' = \Sigma$; $C' = C \cup \{x\}$ and:

- $Inv'(init') = urgent$ and $Inv'(s) = L(s) \wedge x \leq t$ for all $s \in S$.
- E' contains all transitions in E and additionally: for all $i \in I, init' \overset{true:\tau,\{x\}}{\longrightarrow} i$.

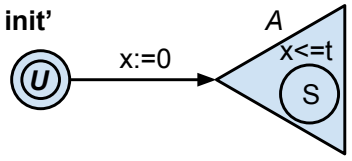

Figure 2.4: Deadline pattern

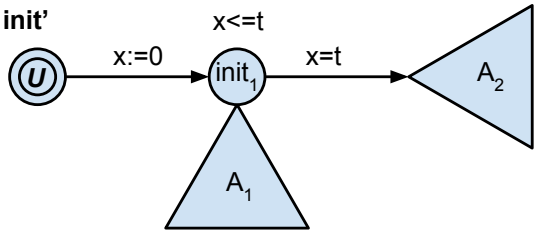

Figure 2.5: Time-out pattern

Time-Out A time-out pattern defines the requirement to start an operation within given time t [156]. Consider the example system in Figure 2.5 and a time-out pattern $A_1 \triangleleft_t A_2$, where $t \in \mathbb{R}_+$. Clock x is reset along the transition from the initial state $init'$. Each initial state $init_1 \in A_1$ is constrained with invariant $x \leq t$. If the control moves out of the initial state, the system behaves according to the A_1 specification. Otherwise, a time-out occurs and A_2 takes over the control.

The example time-out pattern is defined formally as follows. Let A_1 and A_2 be $A_i = (\Sigma_i, S_i, I_i, C_i, Inv_i, E_i)$, where $i \in \{1, 2\}$. Time-out pattern $A_1 \lhd_n A_2$, is then a timed automaton $(\Sigma, S, I, C, Inv, E)$, where:

- $S = S_1 \cup S_2 \cup init'$.
- $I = \{init'\}$.
- $\Sigma = \Sigma_1 \cup \Sigma_2$.
- $C = C_1 \cup C_2 \cup \{x\}$.
- $Inv(init') = urgent$; all A_1 initial states are constrained by time-out invariants – $Inv(s) \wedge x \leq t$ for all $s \in I_1$, whereas A_2 initial states are not – $Inv(s) = Inv_1(s)$
- E contains all transitions in E_1 and E_2 and for all $init_1 \in I_1$ and $init_2 \in I_2$ and $s_1 \in S_1$, $init' \overset{true:\tau,\{x\}}{\longrightarrow} init_1$ and $s_1 \overset{x=t:\tau,\varnothing}{\longrightarrow} init_2$.

2.3.2 UPPAAL

UPPAAL [38] is a toolbox jointly developed by Uppsala University and Aalborg University for the verification of real-time systems. The primary goal was to create an efficient and easy-to-use tool for modelling, simulating and model checking systems in either graphical of textual modelling language. It has been applied successfully in case studies ranging from communication protocols to multimedia applications [137].

The toolbox consists of three main parts: a description language, a simulator, and a model checker [122]. The first part of the description language is based on a non-deterministic guarded command language and the notion of timed automata. The language supports simple data types and extends timed automata with more general data variable types such as Boolean and integer. The second part is the simulator, which allows an interactive graphical examination of the dynamic behaviour of a system and traces, generated by the model checker. Finally, the model checker provides fully automatic means of verifying the model for safety and liveness properties.

UPPAAL property specification language is based on a restricted subset of TCTL (timed computation tree logic) that can check for reachability, safety and liveness properties [38] in the forms:

$$\varphi ::= \forall\,\Box\beta \| \exists\,\Diamond\beta \qquad \beta ::= a \| \beta_1 \wedge \beta_2 \| \not\beta \tag{2.26}$$

Where a is an atomic formula being either an atomic clock (or data) constraint or a component location. Intuitively, for $\forall\,\Box\beta$ to be satisfied all reachable states must satisfy β. Analogously, for $\exists\,\Diamond\beta$ to be satisfied some reachable state must satisfy β.

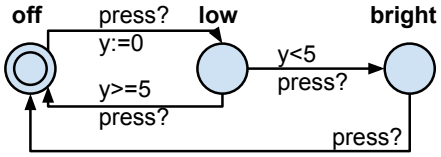

Figure 2.6: Lamp example

Figure 2.7: Lamp user example

UPPAAL extends timed automata with channels and shared variables to support parallel composition. The use of a synchronisation channel is shown in Figure 2.6 as an example.

Figure 2.6 shows a timed automaton modelling a simple lamp in a graphical notation [38]. The lamp has three locations: *off*, *low*, and *bright* and a channel *press*. If the user presses a button (Figure 2.7), *press*! synchronises with *press*?, and the lamp is turned on. If the user presses the button quickly twice, the lamp becomes bright. The user can press the button randomly at any time or even not press the button at all. The clock y of the lamp is used to detect if the user was fast ($y < 5$) or slow ($y >= 5$).

Chapter 3

Event-B: Structure, Refinement and Time Modelling

In this chapter, we introduce the Event-B formalism. We begin by covering its origins, key features and syntax. Then, we demonstrate the formalism in a small traffic light example. Further, we explain that which we relied upon in our case studies. Finally, we overview the relevant work done on modelling timing in Event-B, discuss Sarshogh's discrete timing property approach and give the semantics for it.

3.1 Event-B

The Event-B [19] formalism is an evolution of the Classical B method [18]. Most of the formal concepts it uses were already proposed in Action Systems [28], TLA+ [120], and UNITY [57, 19]. Event-B focuses on reactive systems and applies to hardware and software elements whereas the Classical B is just for software. We prefer Event-B for the simplicity of its notations, extensibility and tool support.

3.1.1 Event-B Modelling

All Event-B models can be described by two basic constructs: *contexts* and *machines*. Contexts specify the static part of a model in forms of carrier sets s, constants c and axioms $A(s, c)$ (Figure 3.1). Context is identified by the *name* and can *extend* a number of other contexts. Machines represent the dynamic part of the model and contain variables v, concrete machine invariants $J(s, c, v)$ and *events* (Figure 3.2). A machine has an identifying name, it may *refine* one abstract machine and can *see* a list of contexts.

CONTEXT *name*
EXTENDS $context\_name_1$, .., $context\_name_n$
SETS s
CONSTANTS c
AXIOMS $A(s, c)$
END

Figure 3.1: Context

MACHINE *name*
REFINES *machine_name*
SEES $context_1$, ..., $context_n$
VARIABLES v
INVARIANTS $J(s, c, v)$
EVENTS *events*
END

Figure 3.2: Machine

An event may accept a number of parameters x and consists of at least two blocks: concrete guards $H(x, s, c, v)$ that describe the conditions need to hold for the occurrence of the event, and actions which determine how specific state variables change as a result of the occurrence of the event (Figure 3.3). The action block takes its meaning from a set of Before-After Predicate $BAP(x, s, c, v, v')$ that modifies the model state.

Event *name* $\widehat{=}$
any x
where $G(x, s, c, v)$
then $BAP(x, s, c, v, v')$
end

Figure 3.3: Event

Contexts can be extended by other contexts and machines can be refined by other machines. Each machine may refer to one or more contexts.

3.2 Refinement

Refinement is the principal feature of Event-B. The method helps a modeller to identify the central problem or functionality of the system and then gradually add in other features thereof. It improves the traceability of requirements as specification details are introduced in small steps. Abrial, the author of Event-B, describes two kinds of refinements for the Event-B modelling [19]. We are going to overview them in the following subsections.

3.2.1 Superposition refinement.

The feature augmentation refinement, also known as *horizontal refinement*, introduces new features of the system. This refinement helps designers to acquire a better understanding by achieving a more precise model in a step-wise manner.

When making a horizontal refinement, we extend the state of a model by adding new variables. The behaviour of these variables and their relation to the abstract functionality is then defined by invariants. The new behaviour is then modelled with new actions that operate on the new variables. The invariants are preserved with the new guards.

The refinement can introduce new events that refine *skip*. These events can modify only new variables and do not affect the abstract level. The horizontal refinement process is completed when all the requirements are specified in the model.

A refinement process generates proof obligations (PO) – a requirement to prove the consistency between the abstract and the concrete refinement levels. We discuss relevant POs in 3.3. Typically, implementability is not considered in a horizontal refinement phase, and the primary focus is the mathematical model that reflects the system requirements.

3.2.2 Data refinement.

Data refinement, also known as *vertical refinement*, enriches the structure of a model to bring it closer to an implementation structure. As a result, we do not enter any more new details of the problem in the model; rather we transform some state and transitions of our discrete system so that it can easily be translated to an implementation language.

When making a vertical refinement, we can remove some variables and add new ones. A typical example of vertical refinement is the transformation of finite sets into Boolean arrays. An important aspect of vertical refinement is the so-called gluing invariant linking the concrete and abstract states. Gluing invariants ensure that the behaviour of the abstract model is preserved with the new variables. In contrast, invariants defined in horizontal refinements mostly define system properties and behaviour.

3.3 Proving

Event-B events generate *proof obligations*. A proof obligation is a mathematical theorem that must be proved to be correct. There are various kinds of proof obligations (POs) concerned with different proof problems. For instance, an Invariant Preservation PO (INV) indicates that the invariant condition is preserved by every event. It is expressed in the following form:

$$\begin{array}{l} A(s,c) \\ J(s,c,v) \\ H(x,s,c,v) \\ BAP(x,s,c,v,v') \\ \vdash \\ i(s,c,v') \end{array} \tag{3.1}$$

where $i(s, c, v')$ is a modified specific invariant. The above proof sequence has two parts which are separated by $\vdash$ sign. The upper part of the sequence is the hypotheses of the proof, and the bottom part is the goal of the proof.

An abstract guard may be removed in the refinement. By doing so, we then have to prove that when the concrete event is enabled, its corresponding abstract event is also enabled. In other words, that the guards of a concrete event are as strong as the guards of its corresponding abstract event. For each abstract guard G_i which has been removed in the concrete refinement, the following proof obligation is generated:

$$\begin{array}{l} A(s,c) \\ I(s,c,v) \\ J(x,s,c,v) \\ H(x,s,c,v,v') \\ W_p \\ \vdash \\ G_i(s,c,v') \end{array} \tag{3.2}$$

Where J is the conjunction of all concrete machines' invariants; H – conjunction of concrete guards. In the above statement W(u,w) is called witness. Witnesses specify the relation between abstract and concrete parameters. Witnesses are similar to local gluing invariants.

3.4 Example Model

Consider an example model of the traffic light system. The abstract machine $m0$ in Figure 3.4 models light controls with Boolean values. When variable *peds_go* is *true*, pedestrians are allowed to pass. Car movement is modelled with variable *peds_go* in the same way. With invariant $inv3$ we express the safety requirement that cars and pedestrians cannot be allowed passage at the same time. Event *set_peds* switches the

pedestrian light. To preserve the safety invariant, condition *grd*2 grants pedestrian passage only if car traffic is forbidden. Machine $m1$ then refines the abstract model with the actual traffic light colours (Figure 3.5). The set *COLOURS* is defined in context $c1$ and represents *Red* and *Green* values. Gluing invariant $gluing_1$ relates abstract variable $peds\_go = TRUE$ to concrete $peds\_colours = Green$. The refined event *set_peds* models pedestrian light in colours. Witness relates the refined away variable v to the new variable c.

MACHINE m0
VARIABLES cars_go peds_go
INVARIANTS
inv1 : cars_go ∈ BOOL
inv2 : peds_go ∈ BOOL
inv3 : ¬ (cars_go = TRUE ∧ peds_go = TRUE)
EVENTS
Initialisation
begin
act1 : cars_go := FALSE
act2 : peds_go := FALSE
end
Event *set_peds* ≙
any v
where
grd1 : v ∈ BOOL
grd2 : v = TRUE ⇒ cars_go = FALSE
then
act1 : peds_go := v
end
Event *set_cars* ≙
any v
where
grd2 : v ∈ BOOL
grd3 : v = TRUE ⇒ peds_go = FALSE
then
act1 : cars_go := v
end
END

Figure 3.4: Abstract machine

MACHINE m1 **REFINES** m0 **SEES** c1
VARIABLES cars_colours peds_colours
INVARIANTS
inv1 : cars_colours ∈ COLOURS
inv2 : peds_colours ∈ COLOURS
$gluing_1$: peds_go = TRUE ⇔ peds_colours = Green
$gluing_2$: cars_go = TRUE ⇔ cars_colours = Green
EVENTS
Initialisation
begin
act3 : cars_colours := Red
act4 : peds_colours := Red
end
Event *set_peds_colours* ≙
refines *set_peds*
any c
where
grd1 : c ∈ COLOURS
grd11 : Green = c ⇒ cars_colours = Red
with
v : v = TRUE ⇔ Green = c
then
act1 : peds_colours := c
end
Event *set_cars_colours* ≙
refines *set_cars*
any c
where
grd1 : c ∈ COLOURS
grd11 : Green = c ⇒ peds_colours = Red
with
v : v = TRUE ⇔ Green = c
then
act1 : cars_colours := c
end
END

Figure 3.5: Refinement machine

3.5 Tooling Support

One of the key advantages of Event-B is its tooling support. Rodin [20] is an Eclipse based IDE for Event-B that provides effective support for modelling, refinement and proof. Rodin auto-generates POs for project machines. These are then discharged by automated theorem provers, such as Atelier-B[1] or SMT [65], or manually via the interactive proving environment.

Rodin provides a wide range of plug-ins, such as Camille text editor, ProB [125] finite model checker. ProB has a temporal and state-based model checker and provides a manual validation of the model using animation. iUML is a UML-like modelling tool that translates the diagrams to Event-B constructs. Due to the extent of the tool's usage in our work, we dedicate the following section to the overview of the tool.

3.5.1 iUML State Machines

iUML plug-in is an integrated graphical front-end for Event-B. The tool consists of diagrammatic class-diagram and state machine modelling notations. It translates drawn diagrams into Event-B code that can then be verified and further augmented manually. It is designed to complement and integrate well into an already existing Event-B code.

We explain the dynamics of the iUML in an example state machine *exampleSM* (Figure 3.6). The state machine is initialised upon firing transition (event) *INITIALISATION*. The action activates the only state *parentS*.

State *parentS* has two sub-state machines, also called regions: $sm1$ at the upper region and $sm2$ at the bottom region. Whenever the parent state is active, sub-state machines must be active as well. This means that upon initialisation $sm1$ and $sm2$ are initialised to states $s1$ and $s3$ respectively.

The concurrent state machines synchronise with shared events. In the given example, such events are $e1$, $e3$ and *exit*. The transition can be fired only when all its outgoing states are active. Transitions $e1$ and $e3$ will not be enabled until all outgoing (source) states are active. Hence, event $e2$ must be executed so that state $s4$ becomes active. When this happens, $e1$ may fire, activating $s2$ and $s5$. At this point, transitions *exit* and $e3$ become enabled because their source states are active. Pseudo-state *Junction* places a condition on a transition for multiple source states to be active. Therefore *exit* is constrained by states $e2$ and $e5$. In contrast, pseudo-state *Any* states that the transition is unconstrained and can be fired at any time. Therefore in our case $e3$ is constrained only by state $e2$.

[1] Atelier-B website: http://www.atelierb.eu/en/

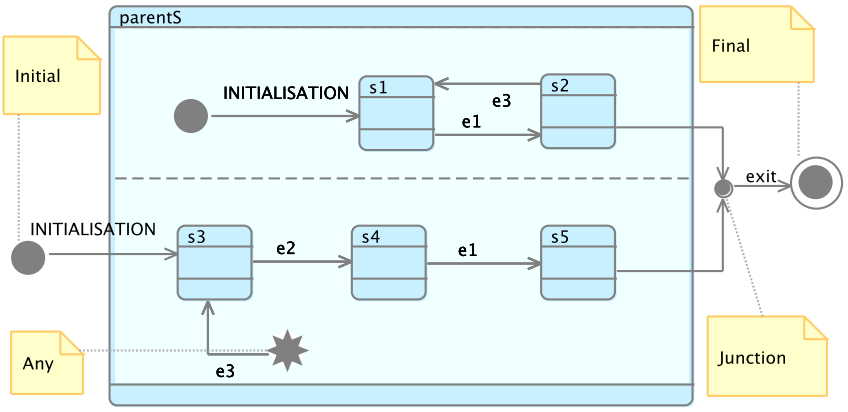

Figure 3.6: iUML state machine *exampleSM*

iUML tool supports two kinds of diagram-to-Event-B translations. We explain the translation kinds on the example state machine (Figure 3.6) in the following paragraphs.

Variable translation. In variable translation, state machine states are represented as Boolean variables, e.g. state *parentS* is represented as variable $s1 \in \mathbb{BOOL}$. When the state is active, the corresponding variable is set to *true*, otherwise – to *false*. Upon initialisation, parent state *parentS*, sub-states $s1$ and $s3$ are set to *true*. When event $e2$ executes, variable $s3$ is set to *false* and $s4$ – to *true*.

A state machine can have multiple instances, each identified by an element from set X. The variables are then declared as subsets of the instance set, e.g. $parentS \subseteq X$. The sub state machine states then use a subset of the active parent state machine indices: $s1 \subseteq parentS$ and $s2 \subseteq parentS$. To ensure that two states of the same machine are not active at the same time, they are partitioned: $partition(parentS, s1, s2)$.

Enumeration and multi-instance. The enumeration translation uses enumerated sets to identify which state in the state machine is currently active. State machines *exampleSM*, $sm1$ and $sm2$ are represented as variables (3.3), (3.4) and (3.4) respectively. E.g., $sm1\_STATES$ has two elements $s1$ and $s2$ that correspond to state machine $sm1$ states.

$$exampleSM \subseteq exampleSM\_STATES \tag{3.3}$$

$$sm1 \subseteq sm1\_STATES \tag{3.4}$$

$$sm2 \subseteq sm2\_STATES \tag{3.5}$$

Their types are sets containing respective state states. Every state machine has a *NULL* state that denotes the fact that the state machine is not active and therefore in none of its states. Note that in contrast to the variable translation, the representation here revolves around the state machine rather than a state itself.

Lifting to a Set of Instances. A state machine can have a set of instances of it running. When lifted, the state machine variable becomes a function, such as (3.6) for state machine $sm1$. Where X is an example instance set.

$$sm1 \in X \rightarrow sm1\_STATES \tag{3.6}$$

An extra parameter is added to each state machine event to represent the contextual instance. In this work the convention is to use a parameter named *self* unless otherwise specified. Consider the dynamics of example event $e1$ with multi-instance support. Guard (3.7) ensures the transition can fire only for such instance *self* which is in state $s1$. Upon execution, the instance's state is updated to state $s2$ (3.8).

$$sm1(slef) = s1 \tag{3.7}$$

$$sm1(self) := s2 \tag{3.8}$$

3.6 Timing in Event-B

Conceptually, events in Event-B are atomic and instantaneous. Event-B lacks explicit support for expressing and verifying timing properties. Modelling time critical systems by using Event-B has been investigated in several studies.

[49] describes an approach to modelling discrete time in Classical B, which is the origin of Event-B. They express current time as a natural number variable and model the time flow with an operation that increments that variable. Time flow operation prevents the progress of time if the current time is equal to the deadline.

[55] model a message passing algorithm in Event-B. The authors model time as a variable $time \in \mathbb{N}$. An event *post_time* adds a new *active time* to a variable $at \subseteq \mathbb{N}$. Active time elements are the future events' activation times ($min(at) > time$) that must be handled by the system. Event *Tick* handles the time flow, where the time progress is limited to the next at element – $min(at)$. Event *process_time* then handles the active time. The paper recommends not introducing timing into the model too early. This avoids unnecessary complexity, especially in terms of proof obligation discharge.

[109] extends Cansell's work on the active time approach. He introduces a finite set of system events *evts* and relates it to a set of activation times: $at \in evts \rightarrow \mathbb{P}(\mathbb{N})$. This improvement allows indexing different sets by a process or a name. To facilitate model-checking, [109] shows an approach to refining an infinite model with absolute timing to a finite model with relative timing and show the equivalence of the two models.

[43], like Rhem, uses the extended version of active times that map a set of events to the future time and adds the support for *bounded inconsistency*. He removes the guard from the *Tick* event to allow time to progress beyond the deadline. Instead, he splits event *process_time* into two cases. One event then handles the case when the active time is handled within expected time boundaries. The other handles the case when *at* has not been handled within a deadline.

[143] categorises timing intervals in terms of delay, expiry and deadline. He introduces a notation to specify these timing properties and provides Event-B semantics for the notation. The notation hides the complexity of encoding timing properties in an Event-B model, thus making timing requirements easier to perceive for the modeller.

A typical timing pattern is a trigger followed by a possible response. Therefore, each of Sarshogh's timing constructs specifies a constraint between a trigger event T and either a response event R or a set of response events $R_1...R_n$:

$$Deadline(T;\ R_1...R_n;\ t) \tag{3.9a}$$

$$Delay(T;\ R;\ t) \tag{3.9b}$$

$$Expiry(T;\ R;\ t) \tag{3.9c}$$

$Deadline(T, R_1...R_n, t)$ means that one and only one of the response events $(R_1..R_n)$ must occur within time t of trigger event T occurring (Figure 3.7). In case of $Delay(T;\ R;\ t)$, the response event cannot occur before time t of trigger event occurring (Figure 3.8). $Expiry(T;\ R;\ t)$ means that the response event cannot occur after time t of trigger event occurring (Figure 3.9).

In principle, Sarshogh's classification of time into delay, deadline and expiry corresponds to timed automata delay, deadline and time-out modelling patterns. However, two significant differences must be pointed out. Firstly, Event-B is a discrete modelling language, hence there is no native support for real numbers as in timed automata. To deal with this, [34] and [33] extended the Event-B formalism with continuous behaviour and developed a hybrid cruise control system case study [33]. [50] presented a theory plug-in that could potentially facilitate extension of the language and required proof rules. Secondly, Sarshogh's patterns can be used in a stepwise refinement modelling, whereas timed automata do not natively support such a feature.

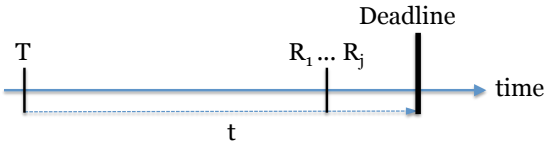

Figure 3.7: Deadline

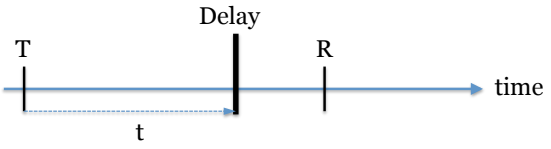

Figure 3.8: Delay

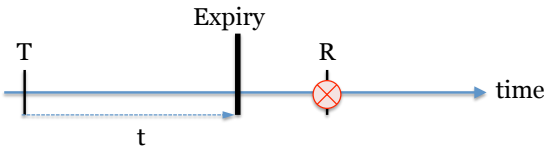

Figure 3.9: Expiry

We chose to investigate Sarshogh's patterns in more detail because of its comprehensive modelling base that includes syntax and time constraint classifications similar to that of timed automata and pattern semantics in Event-B with the refinement support.

3.6.1 Semantics of Sarshogh's Timing Properties

Sarshogh gives semantics to timing constructs by translating them into Event-B variables, invariants, guards and actions. In particular, these timed-Event-B elements constrain the order between trigger event, response events and the time progressing event (*Tick*).

3.6.1.1 Deadline Semantics

We explain the semantics of the deadline timing constraint (3.10) with n ($n > 0$) alternative responses in a generic example (Figure 3.10).

$$Deadline(T;\ R_1, ..., R_n;\ t) \tag{3.10}$$

The deadline of Figure 3.10 is based on the trigger event T, and its possible responses $R_1..R_n$. Variables fT and fR_i, where $i \in n$, hold corresponding event occurrence indices. Variables tT and tR_i are timestamps that associate each occurrence index with its occurrence time in the form $tT : fT \rightarrow \mathbb{N}$. jB_i is an injection from the occurrence of the trigger event to the occurrence of the response event R_i. It specifies the correlation between the indices fT and fR_i. Elements Grd and Act denote other guards and actions depending on constants c, variables v and parameters.

The order of the trigger and response events is specified by invariant inv_1: if the trigger fT has not yet occurred, none of the response events can occur. Invariant inv_2 specifies the property of the current time value, when the trigger event has happened, but no

response event has happened yet. The current time must be within time t of the trigger event occurrence $tT(trig)$. Finally, invariants $inv_3..inv_{n+2}$ express the property that, if a response event R_i happens for a trigger $trig = jR_i^-1(resp_i)$, its occurrence time will not exceed the occurrence time of T for the corresponding parameter by more than t.

Timing constraint related invariants $inv_2..inv_{n+2}$ are preserved with a $g1$ guard in the *Tick* event. The guard ensures that for every triggered event *trig* in fT, which has not yet been responded to by any of the corresponding response events fR_i, the time cannot pass past the deadline duration t from the trigger event occurrence $time \leq tT(trig) + t$.

INVARIANTS

inv_1 : $\forall trig \notin fT \Rightarrow jR_1(trig) \notin fR_1 \land ... \land jR_n(trig) \notin fR_n$

inv_2 : $\forall trig \in fT \land jR_1(trig) \notin fR_1 \land ... \land jR_n(trig) \notin fR_n \Rightarrow time \leq tT(trig) + t$

inv_3 : $\forall trig, resp \cdot resp \in fR_1 \land trig \in jR_1^{-1}(resp) \Rightarrow tR_1(resp) \leq tT(trig) + t$

...

inv_{n+2} : $\forall trig, resp \cdot resp \in fR_n \land trig \in jR_n^{-1}(resp) \Rightarrow tR_n(resp) \leq tT(trig) + t$

Event $T \widehat{=}$
any trig
when
trig $\in$ X
trig $\notin$ fT
Grd_T
then
fT := fT $\cup$ {trig}
tT(trig) := time
Act_T
end

Event $R_i \widehat{=}$
any trig, resp
when
trig $\in$ fT
resp = jR_x(trig)
$jR_1 \notin fR_1$
...
$jR_n \notin fR_n$
Grd_{R_x}
then
$fR_x := fR_x \cup$ {resp}
tR(resp) := time
Act_{R_x}
end

Event *Tick* $\widehat{=}$
any
Tick
when
tick > 0
g1 : $\forall trig \in fT \land jR_1(trig) \notin fR_1 \land ... \land jR_n(trig) \notin fR_n \Rightarrow time + tick \leq tT(trig) + t$
then
time := time + tick
end

Figure 3.10: The semantics of Sarshogh's deadline timing constraint

3.6.1.2 Delay and Expiry Semantics

The delay pattern is constructed based on the same trigger-response principle. Consider a generic delay timing constraint requirement:

$$Delay(T, R, t) \tag{3.11}$$

In the given example in Figure 3.11 invariant inv_1 preserves the order of the trigger and response events. The inv_2 invariant expresses the requirement that response event R occurs after time t has passed from trigger occurrence time $tT(trig)$ - $tR(resp) \geq tT(trig) + t$. To preserve the second invariant, a guard $time \geq tT(resp) + t$ is needed in the event R, current time should be higher than the specified trigger's delay $tR(trig) + t$.

INVARIANTS

inv_1 : $\forall \mathtt{trig} \notin \mathtt{fT} \Rightarrow \mathtt{jR(trig)} \notin \mathtt{fR}$

inv_2 : $\forall \mathtt{trig}, \mathtt{resp} \cdot \mathtt{resp} \in \mathtt{fR} \wedge \mathtt{trig} \in \mathtt{jR}^{-1}(\mathtt{resp}) \Rightarrow \mathtt{tR(resp)} \geq \mathtt{tT(trig)} + \mathtt{t}$

Event $T \widehat{=}$
any trig
when
trig $\in$ X
trig $\notin$ fT
$\mathtt{Grd}_T$
then
fT := fT $\cup$ {trig}
tT(trig) := time
$\mathtt{Act}_T$
end

Event $Tick \widehat{=}$
any Tick
when
tick $>$ 0
then
time := time + tick
end

Event $R \widehat{=}$
any trig, resp
when
trig $\in$ fT
resp = jR(trig)
resp $\notin$ fR
time + tick $\geq$ tT(trig) + t
$\mathtt{Grd}_R$
then
fR := fR $\cup$ {resp}
tR(resp) := time
$\mathtt{Act}_R$
end

Figure 3.11: The semantics of Sarshogh's delay timing constraint

The semantics of the expiry timing constraint is analogous to the delay semantics, except that equality is the opposite. Invariant inv_2 is written as:

$$\forall trig, resp \cdot resp \in fR \wedge trig \in jR^{-1}(resp) \Rightarrow tR(resp) \leq tT(trig) + t \tag{3.12}$$

And the guard on R event is:

$$time + tick \leq tT(trig) + t \tag{3.13}$$

3.7 Refinement of Time in Event-B

To tackle the complexity in a stepwise manner, [143] presents three key refinement pattern types for the timing properties:

- A pattern to refine abstract deadline to sequential sub-deadlines. In Figure 3.12 the given abstract deadline with the duration of t is refined to a sequence of sub-deadlines with durations $t1$ and $t2$, where $t \geq t1 + t2$. The sequence is triggered by tick event T and has to be responded to within time $t1$ by a response event R_1. The time duration between the occurrence of R_1 and R_2 must be at most $t2$.

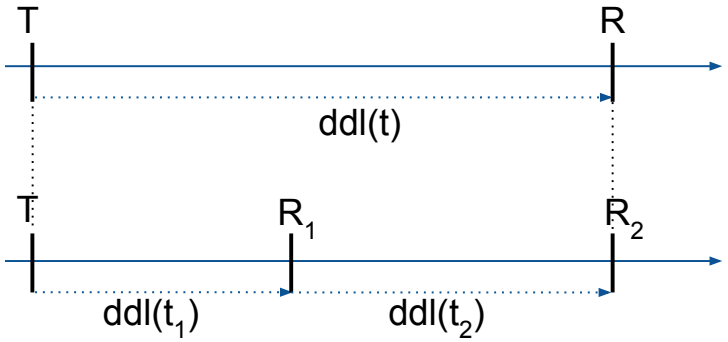

Figure 3.12: Deadline

- A pattern to refine abstract expiry to a sequence of expiry and a deadline.

  In the given example (Figure 3.12) a sequence of expiry and deadline substitute the abstract expiry. Within time $t1$, the sequence can be responded to by event R_1. If the time progresses beyond $t1$ without the response, the response event R_1 expires. Otherwise, event R_1 occurrence must be responded to by event R_2 within time $t2$. Hence, the abstract expiry is broken into a concrete expiry and a concrete deadline in such a way that their sequence does not violate the abstract expiry ($t \geq t1 + t2$).

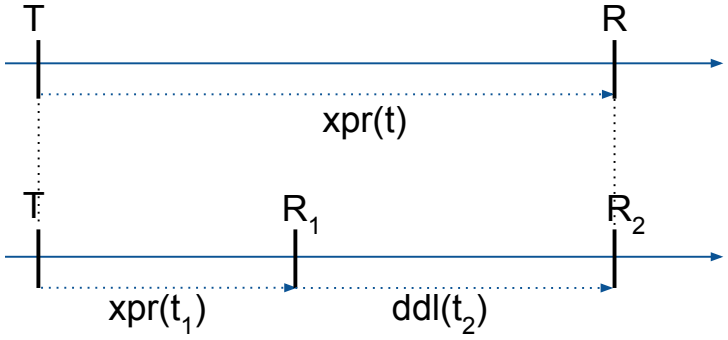

Figure 3.13: Deadline

- The final refinement type refines an abstract response event R to two alternative response events R_1 and R_2. At the concrete level, the trigger must be responded to by either R_1 or R_2 event. within time $t1 = t$.

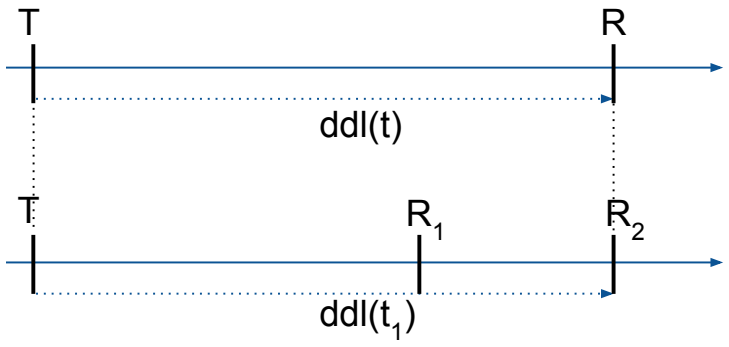

Figure 3.14: Deadline

Other work related to time refinement in Event-B is mentioned in this paragraph. Berthing et al. [107] propose a design workflow with alternating data and timing constraint refinement steps. Stepwise refinement of an Event-B model is guided at each step by transformation to an Uppaal model, which is used for annotation of the next Event-B refinement. The goal of the authors is to take advantage of a refinement support in Event-B, and a verification of timing requirements in UPTA[2]. The automation of the proposed design transformations remains for future work. Méry and Singh [71] apply the approach of [55] to modelling the timing constraints of a single electrode pacemaker system. At the abstract level, the fundamental timing intervals are introduced, then gradually enriched through four refinements. The enrichment is case-specific and is coupled with the model structure, thus limiting the potential for reuse.

[2]UPPAAL lacks a notion of refinement

Chapter 4

Timing Interval Approach

In this chapter, we present our timing interval approach. Firstly, we provide the motivation for the new approach. Secondly, we give a simple example model to highlight the new features of our timing interval. Thirdly, we introduce our timing interval approach through the given example model and then describe the underlying semantics and generative aspect of it. Finally, we overview our approach and discuss the results.

4.1 Motivation

We have chosen Sarshogh's approach to model timing requirements in the pacemaker case study. We prefer Sarshogh's approach for its notation, which hides the complexity of encoding timing properties in an Event-B model. The hidden complexity makes timing requirements easier to express for the modeller. Moreover, the author provides reusable refinement patterns.

We have attempted to model three case studies: a pacemaker (chapter 9), a message passing protocol (chapter 10) and landing gear system (chapter 11). The attempts highlighted the limitations of Sarshogh's approach and led to many case-specific workarounds. The workarounds inspired us to extend Sarshogh's approach with the features discussed below.

The pacemaker case study has shown that some timing constraints can have **dynamic durations**, which are not supported by Sarshogh's approach.

During the modelling process, we identified the need for a **special response type - abort**. Unlike the response event, abort is not constrained by the timing properties and can occur at any point in time. This behaviour is suitable for sensors, where the non-deterministic environment cannot be restricted.

In some cases, a timing property can be triggered by **multiple triggers**. Sarshogh's approach does not define such a feature.

During modelling, we observed **event overloading**. Event overloading occurs when a single event can handle multiple roles of timing constraints, e.g. serve as the trigger and the response at the same time.

Finally, we needed a **higher-level abstraction** that can formally represent multiple related timing requirements as a single entity. The latter would allow reasoning upon thereof and allow elaboration through the refinement of such entities. Also, higher-level abstraction would better map to the iUML state machine models on which we heavily rely.

Although model-checking is not the primary method of verification in our research, it has advantages over deductive verification in terms of temporal logic and enabledness verification. Sarshogh uses an unbounded *time* variable to model time in Event-B. The introduction of such a variable prevents finite-state model checkers such as ProB from potentially covering the full model state-space.

To address the limitations mentioned above in this chapter, we present a timing interval approach that builds on the existing notion of delay, deadline and expiry [143].

We propose the notion of an interval. It is a period of time between the occurrence of a trigger and response events. The interval involves delay and deadline timing properties that constrain the interval's lower and upper duration bounds. An abort event can at any time terminate the interval.

4.2 Simplified Example

We demonstrate the interval approach through an example model. Our development process consists of two main stages. In the first stage, we model the whole system in UML diagrams using the iUML tool. In the second stage, we reiterate through every model refinement and overlay explicit timing using our interval approach. We leverage the power of abstraction and reuse via templates.

The example model is represented in a UML-like diagram that is generated with the iUML tool (Figure 4.1). It is a multi-instance state machine SM with instance set X. The state machine has only one state pm, which has two concurrent regions $sm1$ and $sm2$. A transition is enabled when all its source states are active. Therefore $e3$ is always enabled, $e1$ is enabled when the upper region is in state st_INT1_off. Transition $e2$ works as a synchronisation point – it is enabled only when the upper region is in state st_INT1 and the lower region is in state st_INT2. The regions act independently unless the shared event $e2$ is executed.

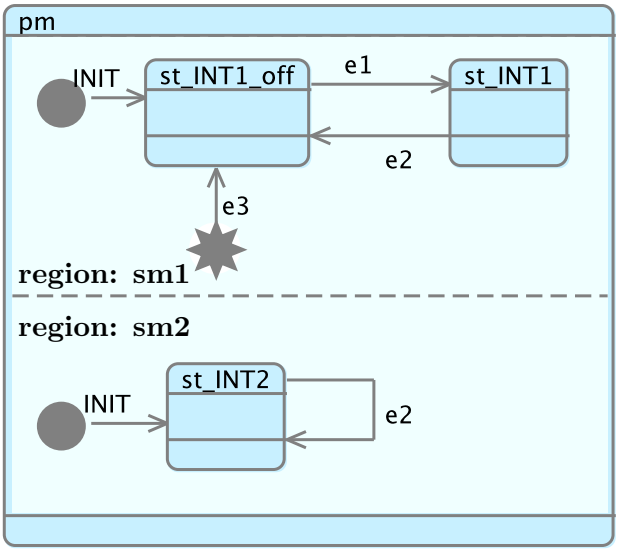

Figure 4.1: Simplified pacemaker example. State machine SM with two regions $sm1$ and $sm2$.

In the upper region $sm1$, we define a graphical interval that is triggered by event $e1$ and responded to by event $e2$. We assume that this interval is an aggregate of delay and deadline timing properties, meaning that the interval has lower and upper duration limits. We call this interval $INT1$. Thus the "interval" is a more abstract conceptual representation that explicitly declares the aggregation fact and formally relates these boundaries together. The aggregate then directly maps to the corresponding state st_INT1. We discuss the recommended convention to represent a graphical interval in a state machine diagram in chapter 6.

We consider the notion of an *abort* event, which can abort an already active timing interval. For instance, at any point in time event, $e3$ must be able to abort the upper region's active timing interval $INT1$. Moreover, we require that the enabledness of event $e3$ is independent of whether $INT1$ is active or not.

The lower region $sm2$ contains a timing interval $INT2$. Interval $INT2$ may be triggered by $INIT$ or $e2$ event; hence, it requires *multiple trigger support* in the same refinement level. Timing interval $INT2$ is responded to with the event $e2$.

Note that the upper and lower-region timing intervals are interdependent – they share event $e2$, which effectively forces a single event to serve as a trigger for the $INT1$ and as both trigger and response for $INT2$. *Event overloading* is the term given to this phenomenon in which an event serves a number of roles in one or more timing intervals.

We explain the dynamics of the intervals in an example execution scenario in Figure 4.2. For simplicity, we display two state machine SM instances, X_1 and X_2, although the

instance number is not limited. Upon model initialisation (1), both state machine instances initialise in states *st_INT*1*_off* and *st_INT*2 for the upper and lower regions respectively. The initialisation immediately triggers two independent interval *INT*2 instances, one for state machine instance X_1 and one for X_2. From this point, the flow of both state machine instances and their corresponding timing intervals runs independently. Therefore, we narrate only the activity of instance X_1. The second occurrence is that of event *e*1 (2) triggering interval *INT*1. Now both intervals are active and run in parallel. At some point, event *e*2 fires (3), responding to *INT*1 and resetting *INT*2. Then, event *e*1 fires (4) triggering *INT*1. This time event *e*3 occurs (5) and aborts *INT*1 before it can be responded to. Event *e*1 follows (6) triggering *INT*1 again. During occurrences of (4), (5) and (6), *INT*2 was not affected and kept running. To highlight the synchrony between the state and the interval, we mark the time periods when state *st_INT*1 is active.

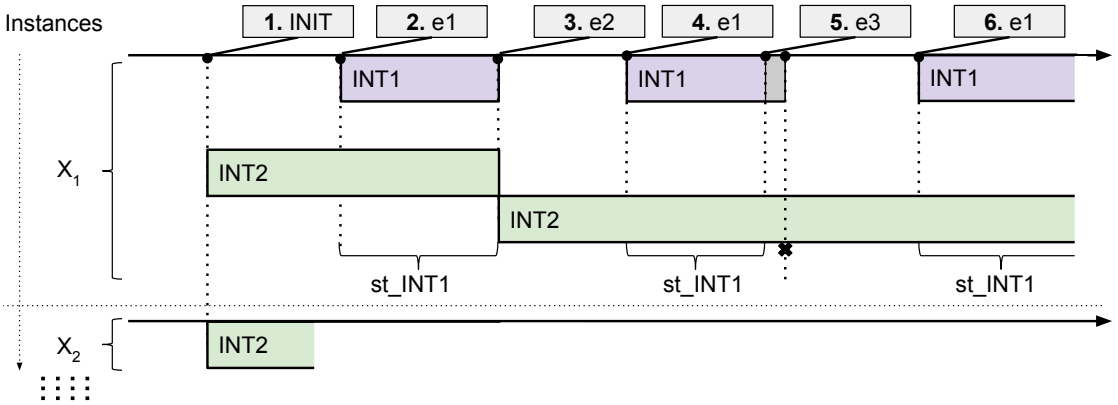

Figure 4.2: Example execution scenario of the multi-instance state machines *sm*1 and *sm*2.

4.3 Timing Interval Notation

To fulfil simplified requirements, we extend Sarshogh's work and introduce the timing interval approach. On top of the deadline, delay and expiry timing properties, we add a higher-level concept of *interval*, extend the timing notation and introduce the *abort* event. We define the interval (4.1) as a modelling abstraction that denotes a period of time. The interval is characterised by a number of parameters: index set, *timing properties TP* and a set of events, including optional ones denoted by [ ]. The system may have a number of timing intervals that are identified by a unique name. Depending on the specified set *IDX*, there may be multiple active instances of a given interval that act independently from each other.

$$\begin{aligned} &Interval(IDX;\ T_1[,...,T_i];\ R_1[,...,R_j];\ [A_1,...,A_k]; \\ &\quad TP_1(TP\_TYPE_1, durType_1(...)) \\ &\quad [,TP_2(TP\_TYPE_2, durType_2(...))]) \end{aligned} \tag{4.1}$$

The interval is manipulated by three kinds of events. One of a set of trigger events $T \in T_1..T_i$ always creates a new instance of the interval. One of a set of response events $R \in R_1...R_j$ always terminates one interval instance under conditions specified by timing properties. If there is no active interval instance to terminate, the response event is disabled. To be well defined, the interval must have at least one trigger and one response event. One of a set of abort events $A \in A_1..A_k$ aborts the interval. Unlike the response event, the abort event is not constrained by timing properties TP and may be enabled if there is no active interval instance to abort. The abort event always aborts an active interval instance (if such exists).

The interval must have either one or two timing properties TP. A timing property takes two parameters. Parameter TP_TYPE is the timing type – *Deadline* (DDL), *Delay* (DLY) or *Expiry* (XPR). Function $durType(...)$ specifies the duration type. Constant duration $C(t)$ takes a parameter t which is either a natural number or a constant name that refers to the actual value. The dynamic duration $D(REF)$ takes one parameter which is an abstract reference to the computed duration. The duration is determined by an algorithm defined manually by the modeller. The parameter is not reflected in the Event-B model and is only used to facilitate the referencing to the duration value. We cover $D(REF)$ in more detail in section 4.6. If more than one timing property is associated with the interval, then there is a relation between the interval's timing property durations (3.9a-3.9c). We allow two combinations: delay with deadline and delay with expiry. The delay duration must be less or equal to the deadline duration ($t_{DLY} \leq t_{DDL}$) and the expiry duration ($t_{DLY} \leq t_{XPR}$). The combination of deadline and expiry does not yield meaningful constraints. Therefore we can specify at most two TPs for an interval.

Timing property duration is specified with integer type due to the lack of Event-B support for real numbers. We circumvent this limitation by changing the interpretation of the integer. For example, a natural number can represent a millisecond, a second or any other unit of time. In this work, we interpret all units of time as milliseconds unless stated otherwise.

Having defined the richer notation, we can now use it to specify the upper region timing constraint $INT1$ (4.2) with trigger $e1$, response $e2$ and abort $e3$. Upon event $e1$ execution, a new interval $INT1$ instance is created with an index identifier drawn from set X. The occurrence of the response event $e2$ then becomes constrained by deadline and delay timing properties whose constant durations are $INT1\_t_{DLY}$ and $INT1\_t_{DDL}$

respectively. The abort event $e3$ can be executed at any given time regardless of the state the model is in. Upon event $e3$ execution, the active $INT1$ instance is aborted (if such exists) and the upper region enters the state st_INT1_off.

$$INT1(X;\ e1;\ e2;\ e3;\ TP_1(DLY, C(INT1\_t_{DLY})), TP_2(DDL, C(INT1\_t_{DDL}))) \quad (4.2)$$

According to the interval $INT2$ specification (4.3), the interval is triggered by $INIT$ or $e2$ events. Event $INIT$ means that the interval is activated immediately upon the model initialisation. The *overloaded* $e2$ event serves as both trigger and response for the interval $INT2$. Therefore, when executed, event $e2$ responds to an already existing interval instance and initiates a new one. The delay timing property means that event $e2$ must occur after time $INT2\_t_{DLY}$ of the trigger event occurring. $INT2$ has no abort, it can therefore be responded to only by the response event $e2$. In this case of a self-looping interval, we use a complex instance index $X \times \{0, 1\}$, which will be explained in subsection 4.4.1.

$$INT2(X \times \{0,1\}, INIT, e2;\ e2;\ --;\ TP_1(DLY, C(INT2\_t_{DLY}))) \quad (4.3)$$

As mentioned before, event $e2$ is an *overloaded* event – it is a response event for $INT1$ and $INT2$ intervals. Therefore $e2$ is constrained by interval $INT1$ and $INT2$ timing properties.

Event $e2$ must occur after the delay time $INT2\_t_{DLY}$ of one of the interval $INT2$ triggers occurring (4.4). Where $t_{INT2\_triggers}$ represents the execution timestamp of either trigger event $INIT$ or $e2$. The timestamp t'_{e2} denotes the $e2$ occurrence that responds to the interval instance after one of the triggers has been executed.

$$t'_{e2} \geq t_{INT2\_triggers} + INT2\_t_{DLY} \quad (4.4)$$

Furthermore, event $e2$ must respond to an active interval $INT1$ instance no sooner than the delay time $INT1\_t_{DLY}$ (4.5) and within the deadline time $INT1\_t_{DDL}$ (4.6) of trigger $e1$ occurring at time t_{e1}. Axiom (4.7) ensures the duration of the delay TP is less or equal to that of the deadline's.

$$t'_{e2} \geq t_{e1} + INT1\_t_{DLY} \quad (4.5) \qquad t'_{e2} \leq t_{e1} + INT1\_t_{DDL} \quad (4.6)$$

$$INT1\_t_{DLY} \leq INT1\_t_{DDL} \quad (4.7)$$

When an event serves as a response for multiple timing intervals, it is enabled only when all of those timing intervals are active and satisfy the relevant timing properties. In this example, event $e2$ can fire when only when $INT1$ is active and its timing delay property

requirement is satisfied. Note that interval $INT2$ is always active. We investigate the use cases of overloading in chapter 6.

4.4 Semantics of Interval

Our timing interval approach relies on the already existing Event-B infrastructure. At first, the functional Event-B model, without explicit time, is created starting from the most abstract machine to the very last refinement. Then, we reiterate through all of the machines starting from the most abstract one. At each machine, we perform two steps. First, we create a timing interval specification using the existing elements such as sets and events. Second, we generate the corresponding timing intervals from the specification. In this section, we will overview the result of the TI generation. The generation process itself is described in the next section. It is expected that all the events specified in the timing interval specification are already present in the model.

In this particular example we have just one machine, with the functional aspect generated with the iUML tool. On top of that, we have specified two intervals, $INT1$ and $INT2$. We give semantics to our interval construct by translating it to Event-B variables, invariants, guards and actions. The interval timing notation serves as a blueprint, indicating the required Event-B constructs and its location in the model. The translation approach is analogous to both $INT1$ and $INT2$ cases unless explicitly specified.

Interval We translate the interval $INT1$ to a set of variables that store the information about interval instances (Figure 4.3). Interval $INT1$-specific variables are prefixed with $INT1\_$. Variable $INT1\_trig$ stores the indices of triggered interval $INT1$ instances. When the interval instance is responded to, its index is copied to the $INT1\_resp$ variable. Trigger and response occurrences are timestamped, and the timestamps are stored in $INT1\_trig\_ts$ and $INT1\_resp\_ts$ variables respectively. We model timestamps as a total function $X \rightarrow \mathbb{N}$, where the index set X serves as a unique identification for the interval instance. In case the interval is aborted, its index is copied to variable $INT1\_abrt$. Each interval instance has a dedicated clock. Function $INT1\_clocks$ records the time elapsed from the start of an interval instance, as long as it is active. To ensure that the time flow is synchronised across multiple timing intervals, we allow time progress in a single event *Tick* which is documented in later paragraphs. Consistency invariant ($INT1\_resp\_INT1\_abrt$) states that the interval instance can be either responded to or aborted, but not both (Figure 4.3).

Timing Properties In Event-B semantics, the timing property is expressed as a set of invariants (Figure 4.4). According to the $INT1$ specification (4.2), the interval

$$
\begin{array}{l}
\texttt{INT1\_trig\_type} : \texttt{INT1\_trig} \subseteq \texttt{X} \\
\texttt{INT1\_trig\_ts\_type} : \texttt{INT1\_trig\_ts} \in \texttt{INT1\_trig} \rightarrow \mathbb{N} \\
\texttt{INT1\_resp\_type} : \texttt{INT1\_resp} \subseteq \texttt{INT1\_trig} \\
\texttt{INT1\_resp\_ts\_type} : \texttt{INT1\_resp\_ts} \in \texttt{INT1\_resp} \rightarrow \mathbb{N} \\
\texttt{INT1\_abrt\_type} : \texttt{INT1\_abrt} \subseteq \texttt{INT1\_trig} \\
\texttt{INT1\_clocks\_type} : \texttt{INT1\_clocks} \in \texttt{INT1\_trig} \rightarrow \mathbb{N} \\
\texttt{INT1\_resp\_INT1\_abrt} : \texttt{INT1\_resp} \cap \texttt{INT1\_abrt} = \varnothing
\end{array}
$$

Figure 4.3: $INT1$ variables

is constrained by two timing properties: the delay and the deadline. The deadline timing property consists of two invariants. The first invariant $INT1\_ddl\_1$ expresses the requirement that while the active interval instance has not yet been responded to or aborted, it must not exceed the deadline duration $INT1\_t_{DDL}$. The second deadline invariant $INT1\_ddl\_2$ requires the active interval $INT1$ instance to be responded to within $INT1\_t_{DDL}$ of the trigger event occurring.

$$
\begin{array}{l}
\texttt{INT1\_ddl\_1} : \forall \texttt{idx} \cdot \texttt{idx} \in \texttt{INT1\_trig} \land \texttt{idx} \notin \texttt{INT1\_resp} \cup \texttt{INT1\_abrt} \Rightarrow \\
\quad \texttt{INT1\_clocks(idx)} \leq \texttt{INT1\_trig\_ts(idx)} + \texttt{INT1\_t}_{\texttt{DDL}} \\
\texttt{INT1\_ddl\_2} : \forall \texttt{idx} \cdot \texttt{idx} \in \texttt{INT1\_trig} \land \texttt{idx} \in \texttt{INT1\_resp} \Rightarrow \texttt{INT1\_resp\_ts(idx)} \leq \texttt{INT1\_trig\_ts(idx)} + \texttt{INT1\_t}_{\texttt{DDL}} \\
\texttt{INT1\_dly\_1} : \forall \texttt{idx} \cdot \texttt{idx} \in \texttt{INT1\_trig} \land \texttt{idx} \in \texttt{INT1\_resp} \Rightarrow \texttt{INT1\_resp\_ts(idx)} \geq \texttt{INT1\_trig\_ts(idx)} + \texttt{INT1\_t}_{\texttt{DLY}}
\end{array}
$$

Figure 4.4: $INT1$ timing property invariants

In order to preserve $INT1$ deadline timing property invariants, a guard $INT1\_ddl\_grd$ is needed in the *Tick* event to ensure that the time will not progress beyond the active interval's deadline boundaries (Figure 4.8).

The delay timing property of $INT1$ is expressed as one invariant $INT1\_dly\_1$ (Figure 4.4), which requires the response to occur after the delay duration $INT1\_t_{DLY}$ of the trigger event occurring. To preserve the delay invariant, $INT1$ response event $e2$ must be constrained by the guard $INT1\_dly\_grd$ (Figure 4.6). The guard ensures that the response event $e2$ is enabled only when the time $INT1\_t_{DLY}$ has passed of the trigger event occurring. The delay timing property does not place constraints on the *Tick* event. Note that if an interval has only the delay timing property, its instance clocks are not bounded until the instance is responded to or aborted. The expiry timing property is similar to that of the delay. Hence, we provide only the generic version thereof in section 4.5.

Event *Tick* models the time flow for both intervals $INT1$ and $INT2$ (Figure 4.8). Action $INT1\_clck\_act1$ increments all active timing interval $INT1$ instance clocks. The responded to and aborted instances become irrelevant in terms of timing requirements and thus their clocks are excluded from the time flow modelling.

As described in the previous paragraphs, the $INT1$ deadline timing property places a guard in the *Tick* event. The guard limits the interval's clock values to a maximum of $INT1\_t_{DDL}$. Interval $INT2$ does not have a deadline timing property, therefore $INT2\_clocks$ values are unbounded. Unbounded variables make it impossible to fully cover the whole state-space with finite-state model checkers such as ProB. To tackle the issue, we add an additional condition to action $INT2\_clck\_act1$ (4.8). The condition is marked in bold. This stops the clock's progress if it has reached the minimum required delay value $INT2\_t_{DLY}$:

$$\begin{aligned} &INT2\_clocks := INT2\_clocks \leftarrow (\lambda\, idx \cdot idx \in INT2\_trig \setminus (INT2\_resp \cup INT2\_abrt) \wedge \\ &\quad \mathbf{INT2\_clocks(idx) \leq INT2\_trig\_ts(idx) + INT2\_t_{DLY} \mid INT2\_clocks(idx) + 1)} \\ &\quad INT2\_clocks(idx) \leq INT2\_trig\_ts(idx) + INT2\_t_{DLY} \end{aligned} \tag{4.8}$$

In our approach, the clock's progress beyond the delay duration in a delay-only interval has no benefit.

Events The specified timing notation determines what roles model events take. According to the $INV1$ specification (4.2), event $e1$ serves as the trigger for $INT1$ (Figure 4.5). To trigger a new instance of the interval, an event accepts two parameters. Parameter p_INT1_trig must be an index that has not yet been used ($INT1\_trig\_grd1-2$). Parameter $p\_INT1\_trig\_ts$ sets the interval clock's start time to 0 ($INT1\_trig\_grd3$). At this stage, the use of the second parameter might seem redundant. However, we do this to retain the syntactical consistency in the refinement patterns. The significance of the parameter is discussed in subsection 5.1.4. Upon executing the trigger event, the new index and the timestamp are added to the $INT1$ trigger and timestamp sets ($INT1\_trig\_act1-2$) and the new clock for the specific instance is initialised ($INT1\_trig\_act3$). *Grds* represents the other guards, and *Acts* represents the other actions of the corresponding event.

Event $e2$ serves as the response to $INT1$ (Figure 4.6). The event takes a parameter p_INT1_resp that must be an already existing interval $INT1$ index and has not yet been responded to or aborted ($INT1\_resp\_grd1-2$). Since $INT1$ is constrained by the delay timing property, guard $INT1\_dly$ ensures that event $e2$ is executed no earlier than the duration $INT1\_t_{DLY}$ counting from its execution timestamp $INT1\_trig\_ts$. If the precondition is met, the selected index is recorded into responded event set $INT1\_resp$ with its timestamp.

Event $e2$ is an example of the overloaded event. It serves as the response for $INT1$ and as both the trigger and the response for $INT2$ (Figure 4.6). p_INT2_trig is

a trigger parameter ($INT2\_trig\_grd1-2$) that gets initiated upon event's execution ($INT2\_trig\_act1-2$). The associated timestamp parameter $p\_INT2\_trig\_ts$ is set to 0 ($INT2\_trig\_grd3$). p_INT2_resp is the response parameter ($INT2\_resp\_grd1-2$) that responds to an already active interval instance ($INT2\_resp\_act1-2$). As mentioned in section 4.3, the response event must always respond to an active interval instance. We ensure the requirement with guard $INT2\_resp\_grd3$. Hence $e2$ can be executed only when there are active instances of intervals $INT1$ and $INT2$ to respond, otherwise the event is disabled. There is no interference between these three roles as they operate on different variables.

Event $e3$ serves as an abort for $INT1$ (Figure 4.7). Parameter p_INT1_abrt is modelled as a subset of active but non-responded $INT1$ and non-aborted instance indices ($INT1\_abrt\_grd1$). If upon event execution there is no active $INT1$ instance, the parameter becomes equal to $\varnothing$ and the interval's variable is not affected ($INT1\_abrt\_act1$). On the other hand, if there is at least one active interval instance available, the parameter is forced to contain one index ($INT1\_abrt\_grd2$). When no $INT1$ instance is active, guard $INT1\_abrt\_grd2$ returns $TRUE$ and does not disable the event.

We reuse interval instance indices to avoid infinite state-space. The used indices are reset by a generated interval-specific event $INT1\_reset$ (Figure 4.9). The event takes parameter p_INT1_reset which is a set of indices to be reset. The parameter holds all responded and aborted interval instance indices ($INT1\_rst\_grd1$) and must not be empty ($INT1\_rst\_grd2$). Upon execution, the interval-specific variables are cleared of the indices contained by the parameter ($INT1\_rst\_act\_1-6$).

4.4.1 Mapping Timing Interval to State Machine

We assume that the state machine SM of the given example was translated using enumerated set translation (explained in subsection 3.5.1). Region $sm1$ is represented in Event-B as function (4.9) from the instance index to the available state set (4.10).

$$sm1 \in X \rightarrow sm1\_STATES \tag{4.9}$$

$$sm1\_STATES \in \{st\_INT1\_off, st\_INT1, sm1\_NULL\} \tag{4.10}$$

Mapping invariant $i1$ (Figure 4.10) then maps the timing interval to the specific state in region $sm1$. The invariant states that every active timing interval $INT1$ instance must correspond to the active st_INT1 state. Guards must then be introduced into the event to preserve the invariant. An example guard $g1$ ensures the synchrony between the region and the interval in the trigger event. Similar guards are added to other timing

Event *e1* $\widehat{=}$
any p_INT1_trig p_INT1_trig_ts
where
Grds
INT1_trig_grd1 : $p\_INT1\_trig \in X$
INT1_trig_grd2 : $p\_INT1\_trig \notin INT1\_trig$
INT1_trig_grd3 : $p\_INT1\_trig\_ts \in \{0\}$
then
Acts
INT1_trig_act1 : $INT1\_trig := INT1\_trig \cup \{p\_INT1\_trig\}$
INT1_trig_act2 : $INT1\_trig\_ts := INT1\_trig\_ts \lhd\!\!- \{p\_INT1\_trig \mapsto p\_INT1\_trig\_ts\}$
INT1_trig_act3 : $INT1\_clocks := INT1\_clocks \lhd\!\!- \{p\_INT1\_trig \mapsto p\_INT1\_trig\_ts\}$
end

Figure 4.5: Event $e1$

Event *e2* $\widehat{=}$
any p_INT1_resp p_INT2_trig p_INT2_trig_ts p_INT2_resp
where
Grds
INT1_resp_grd1 : $p\_INT1\_resp \subseteq INT1\_trig$
INT1_resp_grd2 : $p\_INT1\_resp \not\subseteq INT1\_resp \cup INT1\_abrt$
INT1_resp_grd3 : $\exists idx \cdot idx \in X \wedge \{idx\} = p\_INT1\_resp$
INT1_dly_grd : $\forall idx \cdot idx \in p\_INT1\_resp \Rightarrow INT1\_clocks(idx) \geq INT1\_trig\_ts(idx) + INT1\_t_{DLY}$
INT2_trig_grd1 : $p\_INT2\_trig \in X \times \{0, 1\}$
INT2_trig_grd2 : $p\_INT2\_trig \notin INT2\_trig$
INT2_trig_grd3 : $p\_INT2\_trig\_ts = 0$
INT2_resp_grd1 : $p\_INT2\_resp \subseteq INT2\_trig$
INT2_resp_grd2 : $p\_INT2\_resp \not\subseteq INT2\_resp \cup INT2\_abrt$
INT2_resp_grd3 : $\exists idx \cdot idx \in X \times \{0, 1\} \wedge \{idx\} = p\_INT2\_resp$
INT2_dly_grd : $\forall idx \cdot idx \in p\_INT2\_resp \Rightarrow INT2\_clocks(idx) \geq INT2\_trig\_ts(idx) + INT2\_t_{DLY}$
then
Acts
INT1_resp_act1 : $INT1\_resp := INT1\_resp \cup p\_INT1\_resp$
INT1_resp_act2 : $INT1\_resp\_ts := INT1\_resp\_ts \lhd\!\!- (p\_INT1\_resp \times INT1\_clocks[p\_INT1\_resp])$
INT2_trig_act1 : $INT2\_trig := INT2\_trig \cup \{p\_INT2\_trig\}$
INT2_trig_act2 : $INT2\_trig\_ts := INT2\_trig\_ts \lhd\!\!- \{p\_INT2\_trig \mapsto p\_INT2\_trig\_ts\}$
INT2_resp_act1 : $INT2\_resp := INT2\_resp \cup p\_INT2\_resp$
INT2_resp_act2 : $INT2\_resp\_ts := INT2\_resp\_ts \lhd\!\!- (p\_INT2\_resp \times INT2\_clocks[p\_INT2\_resp])$
end

Figure 4.6: Event $e2$

Event *e3* $\widehat{=}$
any p_INT1_abrt
where
Grds
INT1_abrt_grd1 : $p\_INT1\_abrt \subseteq INT1\_trig \setminus (INT1\_resp \cup INT1\_abrt)$
INT1_abrt_grd2 : $INT1\_trig \setminus (INT1\_resp \cup INT1\_abrt) \neq \varnothing \Rightarrow$
$(\exists idx \cdot idx \in INT1\_trig \setminus (INT1\_resp \cup INT1\_abrt) \wedge p\_INT1\_abrt = \{idx\})$
then
Acts
INT1_abrt_act1 : $INT1\_abrt := INT1\_abrt \cup p\_INT1\_abrt$
end

Figure 4.7: Event $e3$

Event *Tick* $\widehat{=}$
when
INT1_ddl_grd : $\forall idx \cdot idx \in INT1\_trig \wedge idx \notin INT1\_resp \cup INT1\_abrt \Rightarrow$
$INT1\_clocks(idx) + 1 \leq INT1\_trig\_ts(idx) + INT1\_t_{DDL}$
then
INT1_clck_act1 : $INT1\_clocks := INT1\_clocks \lhd\!\!- (\lambda idx \cdot idx \in INT1\_trig \setminus (INT1\_resp \cup INT1\_abrt) \mid$
$INT1\_clocks(idx) + 1)$
INT2_clck_act1 : $INT2\_clocks := INT2\_clocks \lhd\!\!- (\lambda idx \cdot idx \in INT2\_trig \setminus (INT2\_resp \cup INT2\_abrt) \wedge$
$INT2\_clocks(idx) \leq INT2\_trig\_ts(idx) + INT2\_t_{DLY} \mid INT2\_clocks(idx) + 1)$
end

Figure 4.8: Event *Tick*

Event *INT1_reset* $\widehat{=}$
any p_INT1_reset
where
INT1_rst_grd_1 : p_INT1_reset = INT1_resp ∪ INT1_abrt
INT1_rst_grd_2 : p_INT1_reset ≠ ∅
then
INT1_rst_act_1 : INT1_trig := INT1_trig \ p_INT1_reset
INT1_rst_act_2 : INT1_trig_ts := p_INT1_reset ⩤ INT1_trig_ts
INT1_rst_act_3 : INT1_clocks := p_INT1_reset ⩤ INT1_clocks
INT1_rst_act_4 : INT1_resp := INT1_resp \ p_INT1_reset
INT1_rst_act_5 : INT1_resp_ts := p_INT1_reset ⩤ INT1_resp_ts
INT1_rst_act_6 : INT1_abrt := INT1_abrt \ p_INT1_reset
end

Figure 4.9: $INT1$ reset event

interval events: $e2$ and $e3$. Currently, mapping is done manually and its automation is part of the future work.

i1 : ∀idx·idx ∈ INT1_trig \ (INT1_resp ∪ INT1_abrt) ⇔ int1SM(idx) = st_INT1
g1 : p_INT1_trig = self

Figure 4.10: Mapping $INT1$ to st_INT1

The $sm2$ region has a *self-loop* transition $e2$. A transition is called self-loop when it has matching source and target states. A timing interval mapping to such a state differs. We discuss this special case in section 7.2.

4.4.2 Timing Interval Dynamics

We explain the dynamics of timing intervals through a simple example scenario on a partial state-space graph (Figure 4.11), represented in a Kripke structure. We omit timestamp and clock variables for clarity and display only interval-related trigger, response and abort variables. States $s1$ through to $s5$ each represent a unique state-space configuration of the variables. For every state, we display values of the variables that have been updated by an incoming transition.

We discuss only one state machine instance X_1, which is synchronised with interval $INT1$ instance X_1 and one of interval $INT2$ instances $X_1 \mapsto 0$ or $X_1 \mapsto 1$. The $INT2$ interval is a singleton interval (discussed later in section 4.7), thus only one of the given two indices can be assigned to an active instance.

The model initialises (Figure 4.2) with an active $INT2$ instance (state $s1$). Interval $INT1$-related variables are empty; index $X_1 \mapsto 0$ is present in the $INT2$ trigger variable, indicating that the interval is already active. Event $e1$ triggers interval $INT1$ by adding index X_1 to its trigger variable $INT1\_trig$. At this point, we can take one of the two available paths. One path is to fire the abort event $e3$. The X_1 index is then added

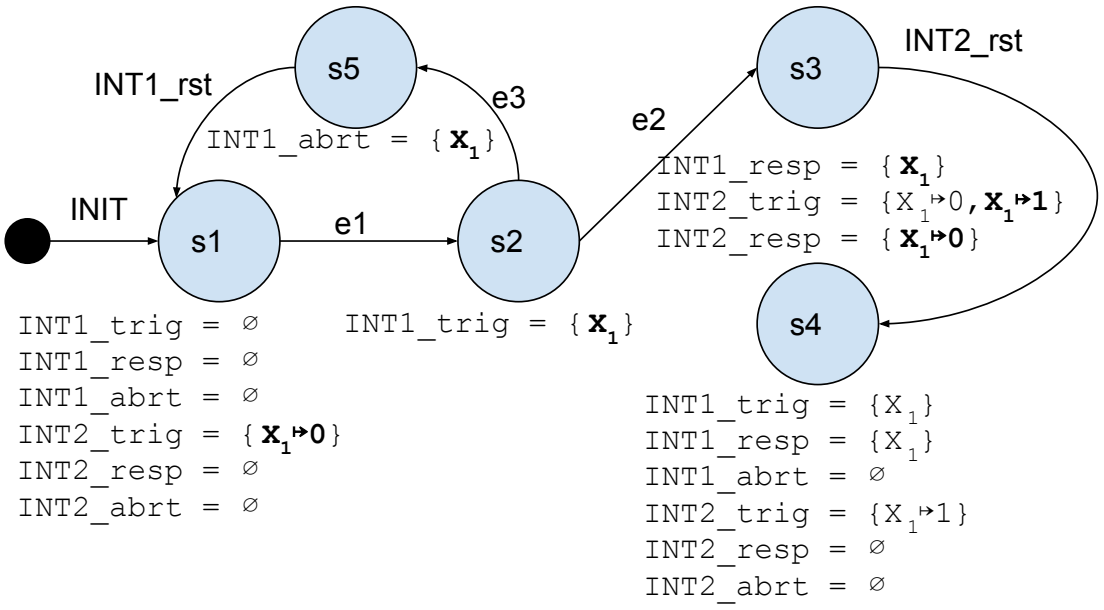

Figure 4.11: The dynamics of $INT1$ and $INT2$ intervals.

to the abort variable $INT1\_abrt$. Event $INT1\_rst$ then removes the index from the variables and we move to state $s1$. The alternative path is to fire the response event $e2$. The event adds active indices of $INT1$ and $INT2$ to the corresponding response variables. Thus, the event triggers a new $INT2$ instance. Reset event $INT2\_rst$ then clears the responded to instance index from interval $INT2$ variables. Note that the event does not affect $INT1$-related variables. The cleared index can then be reused by the next triggered interval $INT2$ instance. Without the reset event, a new interval $INT2$ instance could not be triggered because all of the two indices $X_1 \mapsto 0$ and $X_1 \mapsto 1$ would already be present in the trigger variable.

4.5 Timing Interval Templates

We provide a methodical approach for translating an interval specification to Event-B. The process is based on generic timing interval *templates* that are selectively called by *call statements* depending on the given timing interval specification. The process of generating timing intervals to Event-B consists of three steps (Figure 4.12). We discuss the steps in more detail. Template and call statement definitions are given along the narration.

I. Input. The first step is to construct the timing interval specification (4.1). We assume that in this step all required Event-B events have already been created manually or by a tool such as iUML.

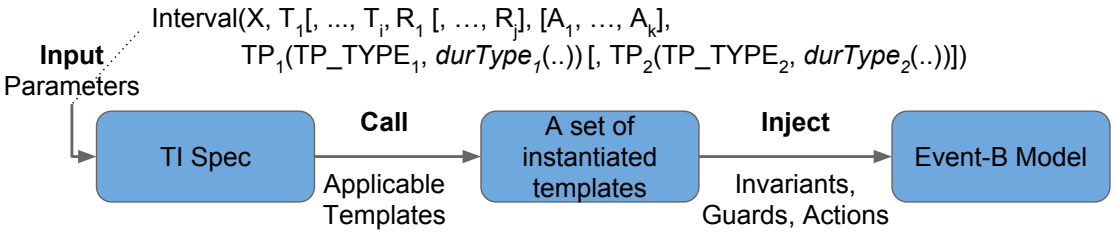

Figure 4.12: Timing interval to Event-B generation workflow.

II. Call. In our approach, templates are not strictly tied to a particular timing interval and can be re-used. We make templates timing-interval specific by calling them with case-specific parameters. This task is up to the call statement. The call statement is a pseudocode expression that may have predicates and contains. If the predicate returns true, a specified template is called with the given parameters. Otherwise, if the expression returns false, the template call is ignored. Each timing interval type has a list of associated call statements.

Consider an example template call DEMO.CALL.1. The antecedent is always true, therefore template *TPL.ROOT.DEMO* is called. Two parameters are passed to the template. Parameter x is a timing interval name for which the template is called. Parameter $\#\mathbf{T(x)}$ is a function that returns a list of trigger events, specified in timing interval x specification. The template call statement starts with *#tc:* .

$$\#tc\text{: } true \Rightarrow TPL.ROOT.DEMO(x, \#\mathbf{T(x)}) \qquad \text{(DEMO.CALL.1)}$$

Template declaration is denoted with the *TEMPLATE* keyword (Figure 4.13). Its name always starts with *TPL*, followed by the timing interval type and the primary function identifiers. In this chapter, we have discussed only one type - *ROOT* - an abstract, non-refining timing interval. Other timing interval types are produced by refinement transformations, covered in chapter 5. There can be many templates primarily (but not exclusively) associated with the same timing interval type. Hence, the function identifier distinguishes templates by their function. In this example, the template's primary function is identified as *DEMO*. Templates may accept a list of untyped, comma-separated parameters. The example template accepts two parameters: a name of timing interval x and a list of event names *evts*. We mark parameters in **bold**.

A template typically contains invariants, guards and actions with generic Event-B elements. When invoked, the templates are instantiated by adding the interval name as a prefix to each generic template variable (remember *INT*1_ and *INT*2_ prefixes in the example project variables). Note the expression given instead of the event name – *ALL e* : *evts*. This expression iterates through all elements in the given list. It says that

TEMPLATE TPL.ROOT.DEMO(X, evts)
@INVARIANTS
i1 : invariants
@Event *ALL e : evts* ≙
@where
g1 : guards
@then
a1 : actions
end

Figure 4.13: Example template call.

every event e in variable *evts* will be injected with the instantiated event code. Some of the templates are nested. The referenced templates are always called and processed before the nested template itself. Templates may have macro commands that perform a specific action before injecting the template into the Event-B model.

III. Inject. Finally, the instantiated template elements are injected into the model. The invariants are injected once; the events are injected into all specified targets.

In the following paragraphs, we explain the generation process for interval $INT1$ and provide relevant templates. A $ROOT$ timing interval type consists of the *interval base template, event templates* and *timing property templates*. We have already completed the first step of specifying the $INT1$ specification (4.2). The second step is to run all call statements associated with type $ROOT$.

Interval Base Template The interval base template is a set of variables and invariants that describe all interval instance states and ensures their consistency (Figure 4.14). Parameter **X** is replaced with the interval name that gets instantiated in the template construct. **@** indicates the target Event-B block to be injected with the instantiated template construct. Figure 4.3 is an instantiation of Figure 4.14. Template element labels of type $X..iN$, $X..gN$ and $X..aN$ mark invariants, guards and actions respectively. They are specific for interval X We leave out the full labels in templates for simplicity and refer to the elements as iN, gN and aN where N is a number.

A single interval base template is required per interval specification. We use the call statement TPL.ROOT.1 to invoke the base template for the current timing interval $INT1$, denoted as parameter x. The second parameter is a function **#idx(x)** that returns the index set used by interval x.

$$\#tc\text{: }TPL.ROOT.Base(x, \#\mathbf{idx(x)}) \qquad \text{(TPL.ROOT.1)}$$

TEMPLATE TPL.ROOT.Base(X, IDX)
@INVARIANTS
X..i1 : $\mathbf{X\_trig} \subseteq \mathbf{IDX}$
X..i2 : $\mathbf{X\_resp} \subseteq \mathbf{X\_trig}$
X..i3 : $\mathbf{X\_abrt} \subseteq \mathbf{X\_trig}$
X..i4 : $\mathbf{X\_trig\_ts} \in \mathbf{X\_trig} \rightarrow \mathbb{N}$
X..i5 : $\mathbf{X\_resp\_ts} \in \mathbf{X\_resp} \rightarrow \mathbb{N}$
X..i6 : $\mathbf{X\_clocks} \subseteq \mathbf{X\_trig}$
consist1 : $\mathbf{X\_abrt} \cap \mathbf{X\_resp} = \varnothing$

Figure 4.14: Interval base template elements.

TEMPLATE TPL.ROOT.RST(X)
@Event $X\_reset \mathrel{\widehat=}$
@any p_X_reset
@where
X..g1 : $\mathrm{p\_X\_reset} = \mathbf{X\_resp} \cup \mathbf{X\_abrt}$
X..g2 : $\mathrm{p\_X\_reset} \neq \varnothing$
@then
X..a1 : $\mathbf{X\_trig} := \mathbf{X\_trig} \setminus \mathrm{p\_X\_reset}$
X..a2 : $\mathbf{X\_trig\_ts} := \mathrm{p\_X\_reset} \lhd \mathbf{X\_trig\_ts}$
X..a3 : $\mathbf{X\_clocks} := \mathrm{p\_X\_reset} \lhd \mathbf{X\_clocks}$
X..a4 : $\mathbf{X\_resp} := \mathbf{X\_resp} \setminus \mathrm{p\_X\_reset}$
X..a5 : $\mathbf{X\_resp\_ts} := \mathrm{p\_X\_reset} \lhd \mathbf{X\_resp\_ts}$
X..a6 : $\mathbf{X\_abrt} := \mathbf{X\_abrt} \setminus \mathrm{p\_X\_reset}$
end

Figure 4.15: $INT1$ reset event

TEMPLATE TPL.ROOT.DDL(X)
@INVARIANTS
X..i1 : $\forall \mathrm{idx} \cdot \mathrm{idx} \in \mathbf{X\_trig} \land \mathrm{idx} \notin \mathbf{X\_resp} \cup \mathbf{X\_abrt} \Rightarrow \mathbf{X\_clocks}(\mathrm{idx}) \leq \mathbf{X\_trig\_ts}(\mathrm{idx}) + \mathbf{X\_t\_DDL}$
X..i2 : $\forall \mathrm{idx} \cdot \mathrm{idx} \in \mathbf{X\_trig} \land \mathrm{idx} \in \mathbf{X\_resp} \Rightarrow \mathbf{X\_resp\_ts}(\mathrm{idx}) \leq \mathbf{X\_trig\_ts}(\mathrm{idx}) + \mathbf{X\_t\_DDL}$
@Event $Tick \mathrel{\widehat=}$
@where
X..g1 : $\forall \mathrm{idx} \cdot \mathrm{idx} \in \mathbf{X\_trig} \land \mathrm{idx} \notin \mathbf{X\_resp} \cup \mathbf{X\_abrt} \Rightarrow \mathbf{X\_clocks}(\mathrm{idx}) + 1 \leq \mathbf{X\_trig\_ts}(\mathrm{idx}) + \mathbf{X\_t\_DDL}$
end

Figure 4.16: Deadline template.

TEMPLATE TPL.ROOT.DLY(X, respEvts)
@INVARIANTS
X..i1 : $\forall \mathrm{idx} \cdot \mathrm{idx} \in \mathbf{X\_trig} \land \mathrm{idx} \in \mathbf{X\_resp} \Rightarrow \mathbf{X\_resp\_ts}(\mathrm{idx}) \geq \mathbf{X\_trig\_ts}(\mathrm{idx}) + \mathbf{X\_t\_DLY}$
@Event $ALL\ e : \mathbf{respEvts} \mathrel{\widehat=}$
@where
X..g1 : $\forall \mathrm{idx} \cdot \mathrm{idx} \in \#\mathbf{getParam(X, resp)} \Rightarrow \mathbf{X\_clocks}(\mathrm{idx}) \geq \mathbf{X\_trig\_ts}(\mathrm{idx}) + \mathbf{X\_t\_DLY}$
end

Figure 4.17: Delay template.

TEMPLATE TPL.ROOT.XPR(X, respEvts)
@INVARIANTS
X..i1 : $\forall \mathrm{idx} \cdot \mathrm{idx} \in \mathbf{X\_trig} \land \mathrm{idx} \in \mathbf{X\_resp} \Rightarrow \mathbf{X\_resp\_ts}(\mathrm{idx}) \leq \mathbf{X\_trig\_ts}(\mathrm{idx}) + \mathbf{X\_t\_XPR}$
@Event $ALL\ e : \mathbf{respEvts} \mathrel{\widehat=}$
@where
X..g1 : $\forall \mathrm{idx} \cdot \mathrm{idx} \in \#\mathbf{getParam(X, resp)} \Rightarrow \mathbf{X\_clocks}(\mathrm{idx}) \leq \mathbf{X\_trig\_ts}(\mathrm{idx}) + \mathbf{X\_t\_XPR}$
@end

Figure 4.18: Expiry template.

TEMPLATE TPL.ROOT.DLY_DDL(X)
@AXIOMS
X..ax1 : $\mathbf{X\_t\_DLY} \leq \mathbf{X\_t\_DDL}$

Figure 4.19: DLY-DDL consist.

TEMPLATE TPL.ROOT.DLY_XPR(X)
@AXIOMS
X..ax1 : $\mathbf{X\_t\_DLY} \leq \mathbf{X\_t\_XPR}$

Figure 4.20: DLY-XPR consist.

Timing Property Templates We define timing property templates for the deadline, delay and expiry. The timing property template is a collection of invariants and guards appropriate for the timing property.

The deadline timing property template consists of two invariants and a guard in the

Tick event (Figure 4.16). Invariants $i1$ and $i2$ express the deadline timing property requirement. Guard $g1$ is for *Tick* event.[1] The template is invoked (TPL.ROOT.2) only if the timing interval has a deadline timing property. We express the condition by checking if the deadline timing property with function *hasDDL*() for a given timing interval.

$$\#tc\text{: } hasDDL(x) = true \Rightarrow TPL.ROOT.DDL(x) \qquad \text{(TPL.ROOT.2)}$$

Similarly, if the timing interval has a delay timing property, the call statement Equation TPL.ROOT.3 invokes the delay template (Figure 4.17). Three parameters are passed to the template. The timing interval x, its response event list **#R(x)** and a reference to the response parameter to use. The delay timing property template consists of a single invariant $i1$ and a guard $g1$ on a response event (Figure 4.17). Note that instead of the event name, we have an expression saying that the following Event-B construct should be injected to all events in variable *evts*. Where *evts* is a template parameter with a response event list. Function **#getParam(X, resp)** returns the response parameter, used by timing interval **X**.

$$\#tc\text{: } hasDLY(x) = true \Rightarrow TPL.ROOT.DLY(x, \#\mathbf{R(x)}) \qquad \text{(TPL.ROOT.3)}$$

The expiry timing property template (Figure 4.18) comprises the same elements as the delay timing property. The only difference is that the interval may be responded to only within time **X**_*t_xpr* from its activation ($i1$). If time progresses beyond **X**_*t_xpr*, guard $g1$ blocks the response event for that particular interval instance. The expiry template call statement (TPL.ROOT.4) is analogous to that of the delay.

$$\#tc\text{: } hasXPR(x) = true \Rightarrow TPL.ROOT.XPR(x, \#\mathbf{R(x)}) \qquad \text{(TPL.ROOT.4)}$$

In case the interval has more than one timing property, their relation must be specified. We give axioms that ensure the consistency of the timing property durations. The deadline and expiry time property durations must always be longer than the delay duration, therefore $\mathbf{X}\_t\_ddl \geq \mathbf{X}\_t\_dly \wedge \mathbf{X}\_t\_xpr \geq \mathbf{X}\_t\_dly$. If the timing interval has delay and deadline timing properties, we call (TPL.ROOT.5) template Figure 4.19. In case of delay and expiry – the template in Figure 4.20 is called (TPL.ROOT.6).

[1] We assume that the time flow event *Tick* are present in the model.

$$\#tc\text{: } hasDLY(x) = true \wedge hasDDL(x) = true \Rightarrow TPL.ROOT.DLY\_DDL(x) \quad \text{(TPL.ROOT.5)}$$

$$\#tc\text{: } hasDLY(x) = true \wedge hasXPR(x) = true \Rightarrow TPL.ROOT.DLY\_XPR(x) \quad \text{(TPL.ROOT.6)}$$

Although we provide patterns for modelling expiry, we do not use it in our case studies. The decision to include the timing property in the timing interval framework was motivated by the fact that the expiry provides a distinct behaviour that cannot be fully replaced with the delay and the deadline. Our intention is to provide a rich framework that could be used to build new timing interval constructs and combinations thereof and be applicable to a wide range of applications.

For each timing interval, we generate a clock variable update action (Figure 4.21). The action is then injected into the *Tick* event. The action updates only those timing interval instances that are not yet responded to or aborted. In case the timing interval has the deadline timing property, the latter then serves as an upper bound for the clock values. The template is called once per timing interval (TPL.ROOT.7).

$$\#tc\text{: } TPL.ROOT.TICK(x, \#\mathbf{idx(x)}) \quad \text{(TPL.ROOT.7)}$$

Interval Event Templates We define Event-B specification templates for trigger (Figure 4.22), response (Figure 4.24) and abort (Figure 4.25) interval event types. Templates consist of parameters, guards and actions that are needed for a specific interval role. The templates are analogous to $INT1$ trigger (Figure 4.5), response (Figure 4.6) and abort (Figure 4.7).

To make the templates specific, we pass to each of them timing interval x and relevant parameters. We pass the interval's index set, trigger event list and a timestamp value to for the trigger template (TPL.ROOT.8); index set and response event list for the response template (TPL.ROOT.9); abort event list for abort event template (TPL.ROOT.10). The instantiated Event-B is then injected into the provided events.

$$\#tc\text{: } TPL.ROOT.T(x, \#\mathbf{idx(x)}, \#\mathbf{T(x)}, 0) \quad \text{(TPL.ROOT.8)}$$

$$\#tc\text{: } TPL.ROOT.R(x, \#\mathbf{idx(x)}, \#\mathbf{R(x)}) \quad \text{(TPL.ROOT.9)}$$

$$\#tc\text{: } TPL.ROOT.A(x, \#\mathbf{A(x)}) \quad \text{(TPL.ROOT.10)}$$

TEMPLATE TPL.ROOT.TICK(X)
@Event *Tick* ≙
@then
X..a1 : $\mathbf{X\_clocks} := \mathbf{X\_clocks} \leftarrow (\lambda\, idx \cdot idx \in \mathbf{X\_trig} \setminus (\mathbf{X\_resp} \cup \mathbf{X\_abrt}) \mid \mathbf{X\_clocks}(idx) + 1)$
end

Figure 4.21: Time progress template

TEMPLATE TPL.ROOT.T(X, IDX, trigEvts, ts)
@Event *ALL* ***e*** : ***trigEvts*** ≙
@any p_X_trig p_X_trig_ts
@where
X..g1 : $p\_\mathbf{X}\_trig \in \mathbf{IDX}$
X..g2 : $p\_\mathbf{X}\_trig \notin \mathbf{X\_trig}$
X..g3 : $p\_\mathbf{X}\_trig\_ts \in \mathbf{ts}$
@then
X..a1 : $\mathbf{X\_trig} := \mathbf{X\_trig} \cup \{p\_\mathbf{X}\_trig\}$
X..a2 : $\mathbf{X\_trig\_ts} := \mathbf{X\_trig\_ts} \leftarrow \{p\_\mathbf{X}\_trig \mapsto p\_\mathbf{X}\_trig\_ts\}$
X..a3 : $\mathbf{X\_clocks} \equiv \mathbf{X\_clocks} \leftarrow \{p\_\mathbf{X}\_trig \mapsto p\_\mathbf{X}\_trig\_ts\}$
end

Figure 4.22: Trigger event template.

TEMPLATE TPL.R(X, respEvts)
@Event *ALL* ***e*** : ***respEvts*** ≙
@any p_X_resp
@where
X..g1 : $p\_\mathbf{X}\_resp \subseteq \mathbf{X\_trig}$
X..g2 : $p\_\mathbf{X}\_resp \not\subseteq \mathbf{X\_resp} \cup \mathbf{X\_abrt}$
@then
X..a1 : $\mathbf{X\_resp} := \mathbf{X\_resp} \cup p\_\mathbf{X}\_resp$
X..a2 : $\mathbf{X\_resp\_ts} := \mathbf{X\_resp\_ts} \leftarrow (p\_\mathbf{X}\_resp \times \mathbf{X\_clocks}[p\_\mathbf{X}\_resp])$
end

Figure 4.23: Generic response event template.

TEMPLATE TPL.ROOT.R(X, IDX, respEvts)
@Event *ALL* ***e*** : ***respEvts*** ≙
#tc: TPL.R(**X**, **e**)
@where
X..g1 : $\exists\, idx \cdot idx \in \mathbf{IDX} \wedge \{idx\} = p\_\mathbf{X}\_resp$
end

Figure 4.24: Response event template.

TEMPLATE TPL.ROOT.A(X, abrtEvts)
@Event *ALL* ***e*** : ***abrtEvts*** ≙
@any p_X_abrt
@where
X..g1 : $p\_\mathbf{X}\_abrt \subseteq \mathbf{X\_trig} \setminus (\mathbf{X\_resp} \cup \mathbf{X\_abrt})$
X..g2 : $\mathbf{X\_trig} \setminus (\mathbf{X\_resp} \cup \mathbf{X\_abrt}) \neq \varnothing \Rightarrow (\exists\, idx \cdot idx \in \mathbf{X\_trig} \setminus (\mathbf{X\_resp} \cup \mathbf{X\_abrt}) \wedge p\_\mathbf{X}\_abrt = \{idx\})$
@then
X..a1 : $\mathbf{X\_abrt} := \mathbf{X\_abrt} \cup \mathbf{X\_abrt}$
end

Figure 4.25: Abort event template.

Note that *TPL.ROOT.R* is a compound template. The instruction within, denoted with *#tc:* , firstly calls and instantiates a generic response template *TPL.R* (Figure 4.23). The generic template provides a general definition of the response parameter and updates the variables. The actual timing interval response template *TPL.ROOT.R* builds on the generic one and adds a specific logic that is relevant to the timing interval – there must be exactly one response index to be responded to. The instantiated template is injected to all events present in parameter *evts*.

4.6 Dynamic Duration

Sometimes the duration of the timing property can be determined only at the time of triggering the associated timing interval. The duration may vary per interval instance, but once set, it stays fixed until the instance is responded to or aborted. We say that such a timing property has a *dynamic duration*.

We explain the dynamic duration with a modified $INT1$ interval specification (4.11). The deadline timing property duration is updated with the dynamic duration function $D(REF)$. The function takes one parameter REF, which serves as an abstract reference to the dynamic duration value determined by an Event-B expression. Typically, such an expression has a case structure to determine the duration value depending on the state of the model and can span multiple lines and is not informative without the context. Hence why it is not displayed in the specification. We provide an example of such an expression in the pacemaker case study in subsection 9.2.2, equation (9.10).

$$INT1(X;\ e1;\ e2;\ e3;\ TP_1(DLY, C(INT1\_t_{DLY})), TP_2(DDL, D(REF))) \tag{4.11}$$

The value of the dynamic duration is determined in the guard block of the trigger event $e1$ and then recorded to the new event parameter $p\_INT1\_DDL\_dur$ (Figure 4.26). The reference parameter REF refers to the new parameter. The parameter is intentionally left undefined and it is up to the modeller to define it. The value is then recorded to the dynamic duration variable $INT1\_DDL\_dur$. The variable is a function that maps interval instance to the duration value:

$$INT1\_DDL\_dur \in INT1\_trig \to \mathbb{N} \tag{4.12}$$

Depending on the Event-B that defines the duration parameter, the duration values may differ for each interval instance.

Constant values in deadline invariants $INT1\_ddl\_1..2$ (Figure 4.4) are replaced with the function $INT1\_DDL\_dur(idx)$ as shown in the modified example in Figure 4.26

The relation between the delay and deadline timing properties is specified as an invariant (4.13). The invariant states that every $INT1$ instance must have a deadline duration that is greater or equal to the specified constant delay duration $INT1\_t_{DLY}$. This invariant replaces the axiom for constant timing property durations (4.7). Similarly we generate consistency invariants for other combinations of constant and dynamic timing properties.

INVARIANTS

INT1_ddl_1 : ∀idx·idx ∈ INT1_trig ∧ idx ∉ INT1_resp ∪ INT1_abrt⇒
INT1_clocks(idx) ≤ INT1_trig_ts(idx) + INT1_DDL_dur(idx)

INT1_ddl_2 : ∀idx·idx ∈ INT1_trig ∧ idx ∈ INT1_resp⇒
INT1_resp_ts(idx) ≤ INT1_trig_ts(idx) + INT1_DDL_dur(idx)

Event *e1* ≙
any Pars p_INT1_DDL_dur
where
Grds
Custom Event-B expression to define p_INT1_DDL_dur
then
Acts
INT1_DDL_dur_act1 : INT1_DDL_dur := INT1_DDL_dur ⩤ {p_INT1_trig ↦ p_INT1_DDL_dur}
end

Figure 4.26: $INT1$ timing property invariants

$$\forall\, idx \cdot idx \in INT1\_trig \Rightarrow INT1\_t_{DLY} \leq INT1\_DDL\_dur(idx) \tag{4.13}$$

Since the dynamic duration values are stored in a variable, they can be modified after they have been set. We explain why this could be problematic with an example instance $X1$ of interval $INT1$. The expected scenario is displayed in Figure 4.27. Upon executing trigger event $e1$, the deadline dynamic duration for $X1$ is set to t_1 time units. The interval instance is eventually responded to with the response event $e2$ after time t_{x1}, where $INT1\_clocks(X1) = t_{x1} \wedge t_{x1} \leq t_1$.

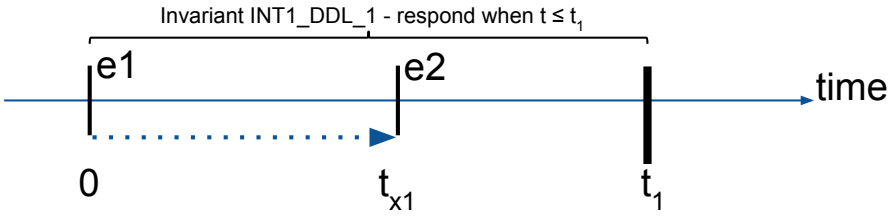

Figure 4.27: Example of triggered $INT1$ insatnce.

Figure 4.28 illustrates the case where the change of the already set dynamic duration value t_1 violates the deadline invariant. We assume that instance $X1$ is active and time t_{x1} progresses to the specified deadline duration t_1. Then, the duration value is changed to t_2, where $t_2 < t_1 \wedge t_2 < t_{x1}$. The new state of the model violates deadline invariant $INT1\_DDL\_1$ (Figure 4.26). The invariant requires that while the active interval instance has not been responded to, the current time t_{x1} should not exceed the deadline duration, which has now been changed to t_2.

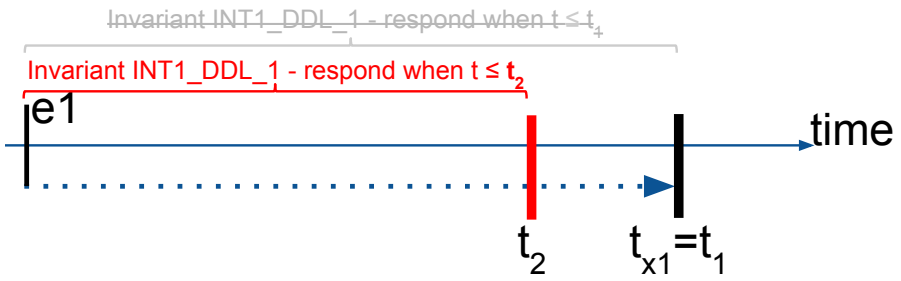

Figure 4.28: Example of invariant violation.

Figure 4.28 illustrates the situation in which the deadline can be pushed back, rendering it meaningless. When time t_{x1} progresses for t_1 time units, the deadline duration is set to t_3 where $t_1 < t_3$. This change does not violate the deadline invariants (Figure 4.26) and allows time t_{x1} to progress further until it reaches t_3. The deadline can be pushed back further indefinitely and the occurrence of the response event $e2$ will never be enforced.

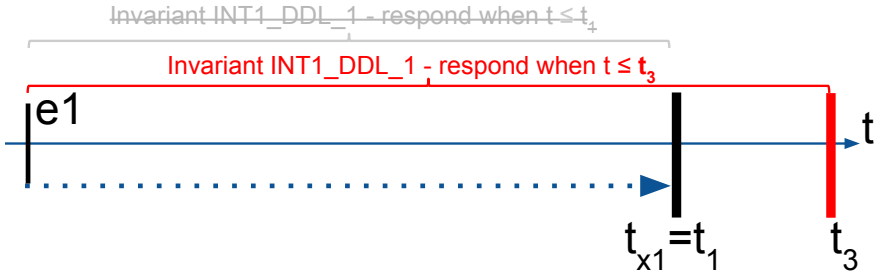

Figure 4.29: Example of the change of the behaviour.

4.7 Singleton Timing Interval

Consider a requirement where an interval can have at most one active instance. We extend timing interval notation to indicate a singleton timing interval and display it on the $INT1$ interval specification (4.14). The index, marked as X_s, denotes that active the interval instance number is limited to at most one at any point in time.

$$INT1(X_s;\ e1;\ e2;\ e3;\ TP_1(DLY, C(INT1\_t_{DLY})), TP_2(DDL, C(INT1\_t_{DDL}))) \tag{4.14}$$

Formally, we ensure this requirement with the invariant $i1$ (Figure 4.30). A guard $g1$ is then added to all trigger events. Overloaded trigger-response and trigger-abort events are omitted because they do not increase the number of active timing interval instances.

We make this template specific with the call statement TPL.ROOT.11, where the second parameter is a pseudocode expression returning all non-overloaded trigger events of timing interval x.

TEMPLATE **TPL.ROOT.Singleton(X, evts)**
@INVARIANTS
X..i1 : $\mathbf{X\_trig} \setminus (\mathbf{X\_resp} \cup \mathbf{X\_abrt}) \neq \varnothing \Rightarrow (\exists \, idx \cdot idx \in \mathbf{X\_trig} \wedge \mathbf{X\_trig} \setminus (\mathbf{X\_resp} \cup \mathbf{X\_abrt}) = \{idx\})$
@Event $ALL\ \boldsymbol{e} : \boldsymbol{evts} \mathrel{\widehat{=}}$
@where
X..g1 : $\mathbf{X\_trig} \setminus (\mathbf{X\_resp} \cup \mathbf{X\_abrt}) = \varnothing$
end

Figure 4.30: Singleton timing interval template.

$$\#tc:\ TPL.ROOT.Singleton(x, \#\mathbf{T(x)} \setminus (\#\mathbf{R(x)} \cup \#\mathbf{A(x)})) \qquad \text{(TPL.ROOT.11)}$$

4.8 Validation and Verification

We have evaluated our timing interval approach regarding verification and validation. The model has two timing intervals ($INT1$ and $INT2$) and 18 time-related invariants. All 69 of the generated timing-related POs were automatically discharged. Verification for deadlock freeness is not well integrated into Event-B framework [163], hence we favour model-checking for this task. To further verify our approach, we fully model-checked our model and did not find any deadlocks or invariant violations.

Moreover, we have written a number of test case scenarios for manual validation with the ProB animator. For this example model, we aimed to validate two aspects: $INT1$ and $INT2$ timing properties against their specifications (4.2, 4.3 respectively) and abort event functionality. In total we have written five test cases that have confirmed that:

- $INT1$ can not be responded sooner than $INT1\_t_{DLY}$ time units;
- $INT1$ is responded within $INT1\_t_{DDL}$ time units;
- $INT2$ cannot be responded sooner that $INT2\_t_{DLY}$ time units;
- $e3$ aborts $INT2$ when only $INT2$ interval is active;
- $e3$ aborts $INT1$ and $INT2$ when both intervals are active.

Potentially, the test cases can be generalised and generated as a set of ProB animation scripts for each specified timing interval. The scripts could then verify if a timing interval can be triggered, responded to and aborted with every or a subset of the specified events. We leave this investigation for future work. We further validate and verify the approach via three case studies (chapter 8).

4.9 Overview and Conclusions

In this chapter, we have used an illustrated example to motivate and introduce the concept of timing interval in terms of state machine and formal timing interval specification.

We have introduced a more detailed notion of timing interval than can be mapped to an iUML state machine both graphically and formally. The guidelines in chapter 6 provide more details on this.

In the simple example model, we have demonstrated some of the features of our approach, such as: the new abort event type and event overloading – when an event can serve many event roles (trigger, response or abort) for multiple intervals. We have demonstrated the Event-B semantics for this example.

We generalised a template-based generative scheme for the transformation of timed models – specified with the state machine and timing interval – to Event-B. The template reusability has been demonstrated in the timing interval response event template. We have developed a tool that automates the timing interval to Event-B generation. The scope of what is automated and what is not is defined in chapter 6.

The approach has been extended to two special cases: singleton and timing properties with dynamic duration. Both features are put to use in the pacemaker case study (chapter 9). The semantics of the constant duration can be considered as an optimisation version of the dynamic duration encoding. It improves the readability of the model by using simple constants rather than custom Event-B expressions in the trigger event guard block. The use of constants generates predictable timing interval constructs, leading to deterministic structure proof obligations. Such proof obligations can then be automatically discharged with our optimised auto prover, discussed in section 8.1. Dynamic duration requires custom Event-B constructs that generate difficult to predict POs. This hinders prover optimisations. Although the optimised auto prover discharged all proof obligations related to the example dynamic duration in the pacemaker case study (chapter 9), we cannot predict how auto-provable the proof obligations are going to be based on undetermined dynamic constraints.

To facilitate the full state-space coverage model-checking we take two steps. Firstly, we have introduced a relative clock for modelling cyclic intervals. Secondly, we have added an index reset method for our approach that clears used interval instance indices.

The generated Event-B specification produces proof obligations that can be automatically discharged by auto provers. Full state-space coverage with finite-state model checkers is possible in the presence of deadline timing property.

We have validated the applicability of our approach in three case studies, discussed in chapter 9, chapter 10 and chapter 11.

Chapter 5

Timing Interval Refinement Patterns

In this chapter, we present five refinement transformations for the timing interval. The goal is to create a methodical infrastructure for modelling timing requirements in a stepwise manner through refinement. The refinement patterns build on our timing interval and follow the same generative approach principle described in chapter 4. We demonstrate each refinement transformation in a small example model and then provide templates for the generative approach. Finally, we investigate the enabledness and deadlock freedom of the timing interval and its refinements.

5.1 Timing Interval Refinement

In this chapter, we present five refinement transformations that allow us to distribute the timing-related requirement complexity while maintaining requirement traceability. The transformations give a modeller a degree of flexibility regarding modelling. We demonstrate the example use of Alternative, Sub-Interval, Abort-to-Response, Single-to-Multi and Retry transformations in two excerpts from the case studies that we have performed. The examples give an idea on how the timing requirements can be elaborated using the refinement transformations.

Figure 5.1 shows a fragment from the pacemaker case study (chapter 9) taking advantage of some of the refinement transformations. In this example, we introduce abstract timing requirement AVI at level $m0$, and then gradually elaborate it with more detail in three refinements. The dynamics of the abstract interval AVI can vary depending on the trigger event that has triggered it. We use the Alternative refinement transformation to split AVI into two alternative intervals $pAVI$ and $sAVI$, each of which is triggered by a different event. The $pAVI$ interval is then transformed into a sequence of sub-intervals

using the Sub-Interval refinement transformation. The occurrence of sub-interval *VSP* abort events is then restricted by transforming them into response events with Abort-to-Response transformation. At each refinement step, the consistency of the transformation is proven.

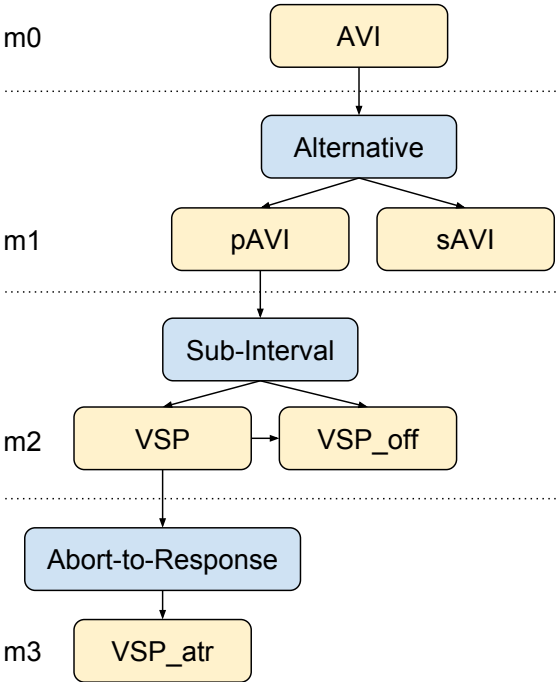

Figure 5.1: Example refinement structure.

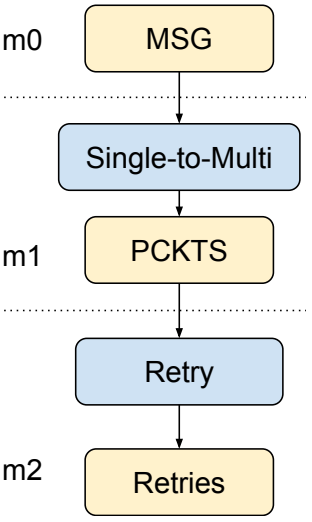

Figure 5.2: Example message transfer refinement structure.

Similarly, in Figure 5.2 we show an excerpt of the message transfer protocol case study (chapter 10). At the abstract level, the message transfer process is represented as a single timing interval *MSG* instance. In the first refinement, we use a Single-to-Multi refinement transformation to refine *MSG* into interval *PCKTS*. The message transfer instance is represented by multiple parallel packet transfer *PCKTS* instances. The packet timing interval *PCKT* is then refined with the Retry refinement transformation into interval *Retries*. The new interval represents each packet transfer instance as a finite sequence of attempts to transfer the packet. The transfer attempt can then be elaborated further. The given refinement transformations provide a methodical and reusable way to split an abstract process into sub-processes.

A timing interval refinement transformation comprises a set of generic Event-B templates. Depending on the timing interval specification and the refinement transformation, specific transformation templates are called by call statements, instantiated and injected into the Event-B model. The templates rely on the already existing events. Therefore, it is up to the modeller to create events that are specified in the timing interval specification.

The refinement process is based on superposition – the abstract timing interval variables are never refined away, but new ones may be added. New invariants define the transformation-specific behaviour and guards that ensure it is respected. Abstract timing property guards are always removed. Their removal produces guard strengthening proof obligations that then must be discharged by adding new invariants and guards. Our approach does not interfere with the functional part of the model and allows data to refine it.

We describe the purpose and working principle for each of the five refinement transformations. We discuss the key invariants and guards involved and explain the generic code templates. We use the example model introduced in chapter 4 as a base for the refinement transformation demonstration. For simplicity, we consider only the upper region with interval $INT1$. Each refinement transformation refines interval $INT1$.

Before explaining the transformations, we overview the $INT1$ timing interval and its variables with a new schematic that we use throughout the refinement pattern demonstration. Figure 5.3 introduces, from left to right, the variables associated with the trigger, abort and response events and the clock variable that stores the interval's progress. When interval $INT1$ is triggered by executing event $e1$ for the first time, a new index IDX from set X is added to the trigger index set variable $INT1\_trig$. Maplet $IDX \mapsto 0$ is recorded in a clock variable $INT1\_clocks$. After the interval is responded to or aborted, the index is added to either index set variable $INT1\_resp$ or $INT1\_abrt$ respectively. Correspondingly, function variables $INT1\_trig\_ts$ and $INT1\_resp\_ts$ record interval instance trigger and response occurrence timestamps. Prior to the firing of a subsequent trigger, the index is cleared from the variables by the index reset event $INT1\_reset$, which is not reflected in the figure. The interval has delay and deadline timing properties with the corresponding durations $INT1\_t_{DLY}$ and $INT1\_t_{DDL}$.

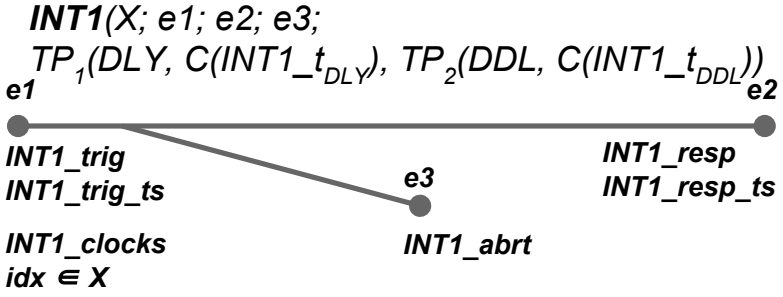

Figure 5.3: Interval $INT1$

We have designed the transformations to be compositional. Transformations always build on the existing timing interval variables listed in Figure 5.3. Variables used by the timing interval may be replaced after the transformation. Moreover, a refinement transformation may introduce auxiliary variables to facilitate the transformation. Such variables are never used by the subsequent refinement transformations. In case the interval has a dynamic timing property duration, the associated variable is included in

the schematic as well. The principle of introducing dynamic durations in the refinement is identical to that which is already covered in section 4.6. For simplicity, we do not consider dynamic duration in this chapter, but demonstrate it in the pacemaker case study in chapter 9.

5.1.1 Alternative Interval Transformation

The alternative refinement transformation refines an abstract interval into two or more alternative intervals. For example, we refine $INT1$ into alternative intervals $ALT1$ and $ALT2$. Only one of the alternative interval instances can be active at a time, which in this example is either $ALT1$ or $ALT2$. four additional requirements apply: (i) alternative intervals $ALT1$ and $ALT2$ cannot share the same trigger or response event; (ii) all trigger and response events must refine their abstract counterparts; (iii) the alternative intervals must have the same timing properties as the abstract timing interval; and, (iv) the property durations must be consistent. The consistency of the timing property durations between intervals $INT1$ and $ALT1$ is ensured with two axioms (5.1) and (5.2), and similarly for $ALT2$. We mark new constants and variables in bold in all further equations.

$$\mathbf{ALT1\_t_{DLY}} \geq INT1\_t_{DLY} \qquad (5.1) \qquad \mathbf{ALT1\_t_{DDL}} \leq INT1\_t_{DDL} \qquad (5.2)$$

Interval $ALT1$ (Figure 5.4) is triggered by event $e11$, whereas interval $ALT2$ – by event $e12$. Both events $e11$ and $e12$ refine the abstract event $e1$. Intervals $ALT1$ and $ALT2$ can be either aborted by $e3$ or responded to by $e21$ and $e22$ respectively. Both response events refine event $e2$.

At the Event-B level, we distinguish alternative interval $ALT1$ and $ALT2$ instances by refining the abstract trigger variable $INT1\_trig$ into new variables $ALT1\_trig$ and $ALT2\_trig$, subject to (5.3). New variables are marked in bold.

$$partition(INT1\_trig, \mathbf{ALT1\_trig}, \mathbf{ALT2\_trig}) \qquad (5.3)$$

The partitioning ensures that the indices are mutually exclusive with respect to alternative intervals $ALT1$ and $ALT2$. Similarly, the abstract response variable $INT1\_resp$ is partitioned to variables $ALT1\_resp$ and $ALT2\_resp$. All new invariants, guards and actions, related to interval $ALT1$, rely on the concrete variables $ALT1\_trig$ and $ALT1\_resp$ instead of the abstract ones (Figure 5.4). These new variables are marked in bold, and reused ones (e.g. $INT1\_abrt$) are in plain text. We will use this convention in all such figures. The dotted line distinguishes diagrammatically between the intervals.

The given code examples for alternative interval $ALT1$, and further refinements as well have been automatically instantiated from the generic code templates by the tiGen tool.

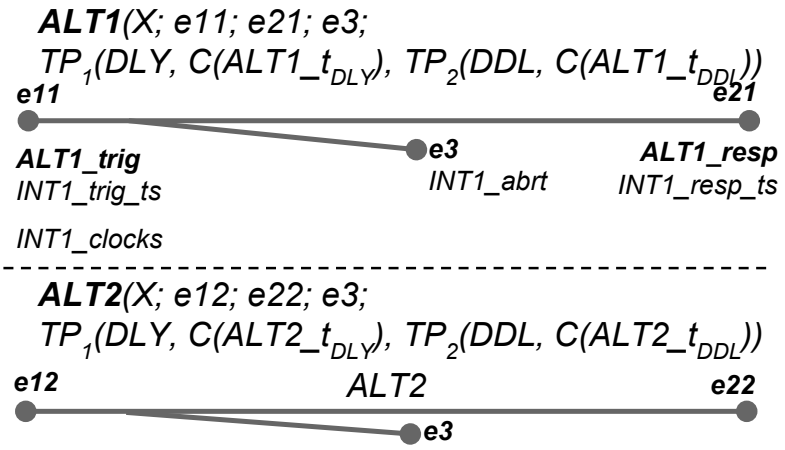

Figure 5.4: Interval $ALT1$

The principle of code generation is analogous to that used to generate timing interval $INT1$ (Figure 5.3). Duration consistency axioms, such as (5.1) and (5.2), are generated similarly to other templates, and thus are not covered. Alternative interval templates have the following naming pattern – $TPL.ALT.*$.

The invariant template (Figure 5.5) defines trigger and response variables for n alternative intervals resulting in code such as (5.3). Parameter **X** is a place holder for the concrete alternative interval name that gets instantiated. **#parent(X)**_*trig* and **#parent(X)**_*resp* are the abstract interval's trigger and response variables. The template is applied once as specified by the call statement TPL.ALT.1. The alternative timing intervals are passed to the template as an array of parameters $x_1, ..., x_n$.

$$\#tc:\ TPL.ALT.Base(x_1, .., x_n) \qquad \text{(TPL.ALT.1)}$$

TEMPLATE **TPL.ALT.Base(X_1, ..., X_n)**
@INVARIANTS
$X_1..i1$: partition(**#parent(X)**_trig, **X_1**_trig, ..., **X_n**_trig)
$X_1..i2$: partition(**#parent(X)**_resp, **X_1**_resp, ..., **X_n**_resp)

Figure 5.5: Alternative: base template.

We define the Event-B code template for the alternative interval trigger event (Figure 5.6). The template is invoked for every alternative timing interval x by call statement TPL.ALT.2. Where $TI_{1..n}$ is a set of refining alternative intervals. The call passes two parameters: the name of the alternative timing interval x and the list of its trigger events **#T(x)**.

$$\#tc:\ \forall x \cdot x \in TI_{1..n} \Rightarrow TPL.ALT.T(x, \#\mathbf{T(x)}) \qquad \text{(TPL.ALT.2)}$$

The template contains one Event-B action. The action adds a new index to the interval's trigger variable **X**_*trig*. Function **#getParam(X, trig)** returns the abstract interval's trigger parameter.

TEMPLATE **TPL.ALT.T(X, evts)**
@Event *ALL e : evts* ≙
@then
X..a1 : **X_trig** := **X_trig** ∪ {**#getParam(X, trig)**}
end

Figure 5.6: Alternative: event templates.

Event *e*11 refines *e*1 ≙
any p_INT1_trig
then
Acts
ALT1_a1 : ALT1_trig := ALT1_trig ∪ {p_INT1_trig}
end

Figure 5.7: Event e11.

Event *e*12 refines *e*1 ≙
any p_INT1_trig
then
Acts
ALT2_a1 : ALT2_trig := ALT2_trigtrig ∪ {p_INT1_trig}
end

Figure 5.8: Event e12.

In this example, we form two calls: (5.4) for the trigger event $e11$ of $ALT1$ with the result displayed in Figure 5.7, and (5.5) for the trigger event $e12$ of $ALT2$ with the final result displayed in Figure 5.8. Function **#getParam(X, trig)** returns abstract trigger parameter p_INT1_trig.

$$\#tc:\ TPL.ALT.T(ALT1, e11) \quad (5.4) \qquad \#tc:\ TPL.ALT.T(ALT2, e12) \quad (5.5)$$

Similarly, call TPL.ALT.3 invokes response event template $TPL.ALT.R$ (Figure 5.9) with two parameters: alternative timing interval x and the list of its response event names **#R(x)**.

$$\#tc:\ \forall x \cdot x \in TI_{1..n} \Rightarrow TPL.ALT.R(x, \#\mathbf{R(x)}) \qquad \text{(TPL.ALT.3)}$$

The invoked template injects a guard and an action to all response events of a specific interval x. The guard ensures that **X**_*trig* contains the index of the timing interval instance to be responded to. The action records the response. Function **#getParam(X, resp)** returns the abstract response parameter's name.

The actual two calls are (5.6) and (5.7) with the result Event-B in (Figure 5.10) and (Figure 5.11) respectively. Function **#getParam(X, resp)** returns the abstract response parameter p_INT1_resp.

$$\#tc:\ TPL.ALT.R(ALT1, e21) \quad (5.6) \qquad \#tc:\ TPL.ALT.R(ALT2, e22) \quad (5.7)$$

TEMPLATE **TPL.ALT.R(X, evts)**
@Event *ALL e : evts* ≙
@where
X..g1 : **#getParam(X, resp)** ⊆ **X_trig**
@then
X..a1 : **X_resp** := **X_resp** ∪ **#getParam(X, resp)**
end

Figure 5.9: Alternative: event templates.

Event *e21* refines *e2* ≙
any p_INT1_resp
where
ALT1_g1 : p_INT1_resp ⊆ ALT1_trig
then
Acts
ALT1_a1 : ALT1_resp := ALT1_resp ∪ p_INT1_resp
end

Figure 5.10: Event e21.

Event *e22* refines *e2* ≙
any p_INT1_resp
where
ALT2_g1 : p_INT1_resp ⊆ ALT2_trig
then
Acts
ALT2_a1 : ALT2_resp := ALT2_resp ∪ p_INT1_resp
end

Figure 5.11: Event e22.

We invoke a template for the deadline timing property for each concrete alternative timing interval (TPL.ALT.4). The call passes just one parameter – alternative interval's name x.

$$\#tc\text{:}\ \forall x \cdot x \in TI_{1..n} \Rightarrow TPL.ALT.DDL(x) \qquad \text{(TPL.ALT.4)}$$

The template (Figure 5.12) calls (*call*1) the previously described deadline template Figure 4.16. Rule #1 in Tick event instructs to remove any existing deadline guard of the abstract interval. The removal of the guard generates the proof obligation that the newly generated deadline guards preserve refinement correctness for the alternative intervals $1..n$.

Figure 5.13 displays the injected template for interval $ALT1$. Note the abstract timestamp, abort $INT1\_abrt$ and clock $INT1\_clocks$ variables are reused as shown in the interval schematics in Figure 5.4. Invariants and a guard in the tick event labelled *call*1 are generated for $ALT1$ by template $TPL.ROOT.DDL$. Label *removed* indicates the removed abstract delay guard for interval $INT1$. Event-B for $ALT2$ is not shown.

Similarly, we invoke the template for the delay timing property (TPL.ALT.5) with two parameters: the alternative timing interval x and the list of its response events R_x.

$$\#tc\text{:}\ \forall x \cdot x \in TI_{1..n} \Rightarrow TPL.ALT.DLY(x, R_x) \qquad \text{(TPL.ALT.5)}$$

TEMPLATE TPL.ALT.DDL(X)
call1 : #TPL.ROOT.DDL(**X**)
@Event *Tick* ≙
#**1:***remove parent DDL guard if such exists*
end

Figure 5.12: Alternative: deadline TP template.

INVARIANTS
Invs
call1 : ∀ idx·idx ∈ ALT1_trig ∧ idx ∉ ALT1_resp ∪ INT1_abrt ⇒ INT1_clocks(idx) ≤ INT1_trig_ts(idx) + ALT1_t_{DDL}
call1 : ∀ idx·idx ∈ ALT1_trig ∧ idx ∈ ALT1_resp ⇒ INT1_resp_ts(idx) ≤ INT1_trig_ts(idx) + ALT1_t_{DDL}
Event *Tick* refines *Tick* ≙
where
Grds
removed : ∀ idx·idx ∈ INT1_trig ∧ idx ∉ INT1_resp∪INT1_abrt⇒INT1_clocks(idx)+1 ≤ INT1_trig_ts(idx)+ INT1_t_{DDL}
call1 : ∀ idx·idx ∈ ALT1_trig ∧ idx ∉ ALT1_resp ∪ INT1_abrt ⇒ INT1_clocks(idx) + 1 ≤ INT1_trig_ts(idx) + ALT1_t_{DDL}
...

Figure 5.13: Instantiated ALT1 deadline template.

The delay template (Figure 5.14) invokes (*call*1) the abstract counterpart Figure 4.17. After the child template is processed, instruction #1 removes abstract delay guards from the response events provided in variable *evts*.

TEMPLATE TPL.ALT.DLY(X, evts)
call1 : #TPL.ROOT.DLY(**X**)
@Event *ALL* ***e*** : ***evts*** ≙
#**1:***remove parent DLY guard if such exists*
end

Figure 5.14: Alternative: delay TP template.

The $ALT1$-specific template call (5.8) generates the delay invariant and a delay guard in interval's response event $e21$. The $INT1$ delay guard is then removed as per template instruction #1.

$$\#tc\text{: } TPL.ALT.DLY(ALT1, e21) \tag{5.8}$$

The newly generated variables for the $ALT1$ interval are cleared in the abstract interval's reset event $INT1\_reset$:

$$\textbf{ALT1\_trig} := \textbf{ALT1\_trig} \setminus p\_INT1\_reset \tag{5.9}$$

$$\textbf{ALT2\_resp} := \textbf{ALT2\_resp} \setminus p\_INT1\_reset \tag{5.10}$$

Where the abstract interval's parameter p_INT1_reset holds all interval $INT1$ indices that are responded to or aborted. The exact semantics of the abstract reset event is displayed in the previous chapter in Figure 4.9. We similarly clear indices from all newly introduced variables in the rest of the refinement transformations unless otherwise stated.

5.1.1.1 Dynamics of ALT1 and ALT2

Figure 5.15 illustrates the dynamics of intervals $ALT1$ and $ALT2$. We display some of the $INT1$-related variables because they are not data refined but rather relied upon by $ALT1$ and $ALT2$ alternative intervals. In this and all further examples, the model initialises in state $s1$ with empty variables. We mark $ALT1$-related variables in **bold**. The newly added variable values are marked in bold as well.

When initialised, we can trigger $ALT1$ by firing $e11$ or $ALT2$ by firing $e12$. We will talk about the former case; the latter is analogous. Upon firing event $e11$, index X_1 is added to the abstract variable $INT1\_trig$ and $ALT1$ trigger variable $ALT1\_trig$ ($s2$).

If the interval instance is responded to by event $e12$, the index is added to the $ALT1$ and $INT1$ response variables ($s3$). Finally, the index X_1 is cleared from the variables by event $INT1\_rst$, and the model returns to the initial state $s1$.

Alternatively, while in state $s2$, active interval $ALT1$ instance can be aborted by firing event $e3$. The index is then recorded to interval's abort variable $INT1\_abrt$. The variable is reused by $ALT1$ as displayed in Figure 5.4.

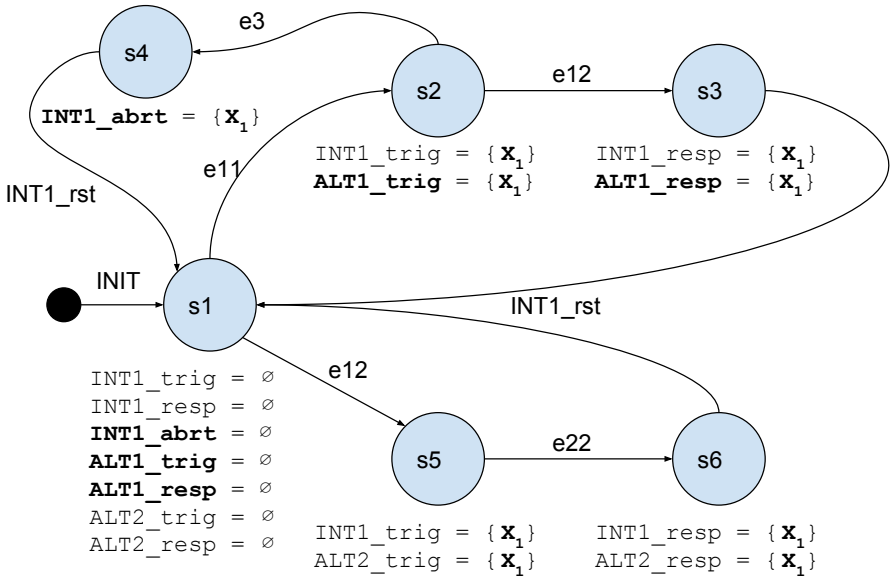

Figure 5.15: Alternative interval $ALT1$ and $ALT2$ dynamics.

5.1.2 Sub-Interval Transformation

The sub-interval refinement transformation refines an abstract interval to a sequence of sub-intervals. We divide interval $INT1$ into a sequence of sub-intervals $SUB1$ and $SUB2$. The order of sub-interval occurrence is strict. Requirements are that (i) the first sub-interval must have the same trigger events as the abstract interval, (ii) the last sub-interval must have the same response events as the abstract interval, (iii) a preceding sub-interval's response event must serve as the trigger event for the succeeding sub-interval, and (iv) the sum of all sub-interval delay durations and the sum of all sub-interval deadline timing property durations must be consistent with those of the abstract interval. For (iv), two axioms are generated: (5.11) ensures that the sum of sub-interval delay durations is greater or equal to that of the abstract interval, and (5.12) is similar for deadline timing property durations.

$$\mathbf{SUB1\_t_{DLY}} + \mathbf{SUB2\_t_{DLY}} \geq INT1\_t_{DLY} \tag{5.11}$$

$$\mathbf{SUB1\_t_{DDL}} + \mathbf{SUB2\_t_{DDL}} \leq INT1\_t_{DDL} \tag{5.12}$$

Sub-interval $SUB1$ (Figure 5.16) is triggered by event $e1$. New event $e4$ serves simultaneously as the response event for interval $SUB1$ and as the trigger event for interval $SUB2$. Interval $SUB2$ is responded to by event $e2$. Both intervals may be aborted by event $e3$. Interval $SUB1$ reuses abstract trigger variables. New variables $SUB1\_resp$ and $SUB1\_resp\_ts$ record the interval's response occurrences. Similarly, variables $SUB2\_trig = SUB1\_resp$ and $SUB2\_trig\_ts$ record $SUB2$ trigger events[1]. Both intervals reuse the abstract abort variable and the index set.

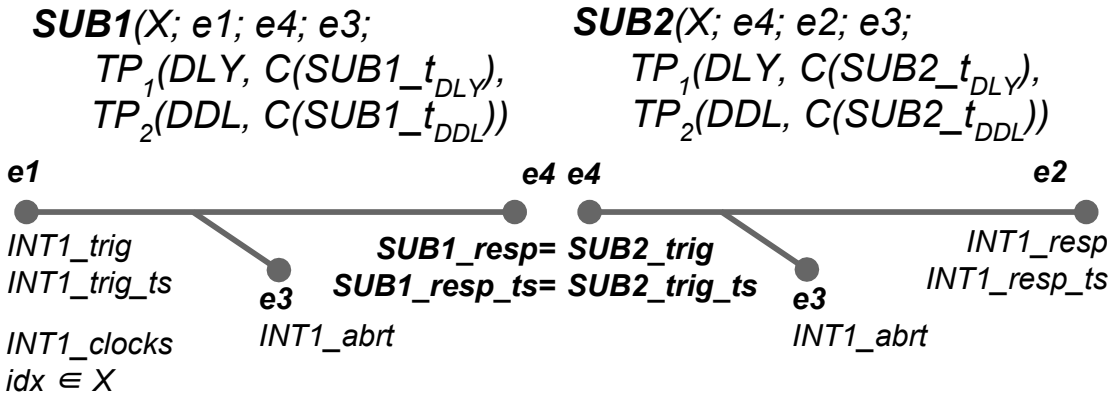

Figure 5.16: Intervals $SUB1$ and $SUB2$

The process of adding delay and deadline timing property invariants for sub-interval $SUB1$ is similar to that of $ALT1$. The abstract clock variable is used to record the

[1] If further sub-intervals are required in sequence, corresponding response and trigger variables will be introduced.

total duration of both sub-intervals. When interval $SUB1$ is responded to, the clock is *not* reset but continues to progress for the duration of the sub-interval sequence. The timestamp, recorded when event $e4$ fires, is then used as an offset to adapt the delay invariant (5.13) for $SUB2$:

$$\begin{aligned} &\forall\, idx \cdot idx \in \mathbf{SUB2\_trig} \wedge idx \in INT1\_resp \Rightarrow \\ &INT1\_resp\_ts(idx) \geq \mathbf{SUB2\_trig\_ts}(idx) + \mathbf{SUB2\_t_{DLY}} \end{aligned} \tag{5.13}$$

We note that syntactically, a sub-interval elaborates a simple interval by (i) overloading response and trigger events at sub-interval junctions, and (ii) running the abstract interval clock for the duration of the sub-interval sequence.

All sub-interval templates are instantiated once for each interval. We define two template functions **#trig(X)** and **#resp(X)** (Figure 5.17) whose value depends on the sub-interval's position in the sequence. Rule #1 says that if it is the first sub-interval, then **#trig(X)** returns the abstract interval trigger variable. If otherwise, it returns the interval-specific trigger variable. Rule #2 says that in case it is the last sub-interval n, **#resp(X)** returns the abstract interval response variable, else it returns the interval-specific response variable.

#1:if x = TI_1 then #trig(X) = #parent(X)_trig, else #trig(X) = X_trig
#2:if x = TI_n then #resp(X) = #parent(X)_resp, else #resp(X) = X_resp

Figure 5.17: Sub-Int: variable rules.

Statement call TPL.SUB.1 invokes a sub-interval base template for all sub-intervals that do not have events serving as abort-triggers. The latter is a special case discussed later in subsubsection 5.1.2.2. Function **#A(x)** returns an abort event list for the given interval x; a composed function call **#T(#succ(x))** returns a set of trigger events of the succeeding sub-interval **#succ(x)**. The intersection of the two function results returns shared abort-trigger events.

$$\#tc\text{:}\ \forall\, x \cdot x \in TI_{1..n} \wedge \mathbf{\#A(x)} \cap \mathbf{\#T(\#succ(x))} = \varnothing \Rightarrow TPL.SUB.Base(x) \tag{TPL.SUB.1}$$

The template (Figure 5.18) declares the relation between trigger and response variables ($i1$) of the specific sub-interval. Condition block $cond1$ – denoted with double square brackets executes – the code block within the curly brackets if the predicate is true. The condition block is true for all intervals X except the last sub-interval T_n, e.g. for interval $SUB1$ the condition is true and reads as $SUB1 \neq SUB2$, where $SUB2$ is the last

sub-interval in the sequence. The block within the curly brackets generates a response timestamp variable ($i2$) and the succeeding sub-interval's trigger variables ($i3$) and ($i4$).

TEMPLATE TPL.SUB.Base(X)
@INVARIANTS
X..i1 : **#resp(X)** $\subseteq$ **#trig(X)**
cond1 : $[[\mathbf{X} \neq \mathtt{T_n}]]${
X..i2 : **#resp(X)**_ts $\in$ **#resp(X)** $\rightarrow \mathbb{N}$
X..i3 : **#trig(#succ(X))** = **#resp(X)**
X..i4 : **#trig(#succ(X))**_ts = **#resp(X)**_ts
}

Figure 5.18: Sub-Int: base template.

INVARIANTS
SUB1_i1 : SUB1_resp $\subseteq$ INT1_trig
SUB1_i2 : SUB1_resp_ts $\in$ SUB1_resp $\rightarrow \mathbb{N}$
SUB1_i3 : SUB2_trig = SUB1_resp
SUB1_i4 : SUB2_trig_ts = SUB1_resp_ts

Figure 5.19: SUB1: base template.

INVARIANTS
SUB2_i1 : INT1_resp $\subseteq$ SUB1_trig

Figure 5.20: SUB2: base template.

Consider the template call statement for interval $SUB1$ (5.14). Variable x is interval $SUB2$, the intersection of its abort variable set ($e3$) and the succeeding sub-interval $SUB2$ trigger variable set ($e4$) return an empty set. Therefore, the antecedent is true and the base template is called for the $SUB1$ interval. The template call for $SUB2$ is constructed in the analogous manner.

$$\#tc\text{: } x = SUB1 \wedge \{e3\} \cap \{e4\} = \varnothing \Rightarrow TPL.SUB.Base(x) \tag{5.14}$$

The instantiated $SUB1$ sub-interval base template (Figure 5.19) declares response variables for $SUB1$ ($SUB1\_i1 - 2$) and trigger variables for $SUB2$ ($SUB1\_i3$-4). The template for $SUB2$ (Figure 5.20) declares only the relationship between $SUB2$ trigger and the response variables ($SUB2\_i1$). We remind the reader that the abstract trigger variable $INT1\_trig$ is reused by $SUB1$, and the abstract response variable $INT1\_resp$ is reused by $SUB2$ (Figure 5.16).

Deadline and delay template calls and the generated result are analogous to that of the alternative interval templates (TPL.ALT.4) and (TPL.ALT.5) respectively.

The sub-interval transformation does not generate any Event-B for the first sub-interval's trigger events. We provide two templates: one for response-trigger events shared by adjacent sub-intervals, and one for the last sub-interval's response events that refine those of the refined interval.

The response-trigger event template *TPL.SUB.RT* is called for all except the last sub-interval n. The template requires three parameters: timing interval x, and its index set and response event list (TPL.SUB.2).

$$\#tc\text{:}\ \forall\, x \cdot x \in TI_{1..(n-1)} \Rightarrow TPL.SUB.RT(x, \#\mathbf{idx(x)}, \#\mathbf{R(x)}) \qquad \text{(TPL.SUB.2)}$$

The template (Figure 5.21) is instantiated and injected to all sub-interval response events **#R(x)**. Note that this is a nested template. It first invokes response template *TPL.ROOT.R* for interval **X** (*call*1). It then generates a gluing guard $g1$, synchronising response and trigger parameters, generated by both child templates. Finally, it invokes trigger template *TPL.ROOT.T* for the succeeding interval **#succ(X)** (*call*2).

Instead of passing an explicit timestamp value to template *TPL.ROOT.T* (the template is displayed in Figure 4.22), we pass a generic expression **#parent(X)**_*clocks*(*p*_**#resp(X)**). The expression is then instantiated within the template depending on the passed parameters. When the template is invoked for sub-interval *SUB*1, function **#parent(X)**_*clocks* in the expression translates to interval's clock variable *INT*1_*clocks* (specified in Figure 5.16). The *p*_**#trig(X)** translates to trigger parameter *p_SUB*2_*trig* for *SUB*2. The semantics of the instantiated expression (5.15) is explained in further paragraphs.

$$INT1\_clocks(p\_SUB2\_trig) \qquad (5.15)$$

TEMPLATE **TPL.SUB.RT(X, IDX, evts)**
@Event *ALL* ***e*** : ***evts*** ≙
@where
call1 : *#tc:* TPL.ROOT.R(**X**, e)
X..g1 : {p_**#trig(#succ(X))**} = p_**#resp(X)**
call2 : *#tc:* TPL.ROOT.T(**#succ(X)**, **IDX**, e, **#parent(X)**_clocks(p_**#trig(X)**))

Figure 5.21: Sub-Int: event templates.

Event *e*4 ≙
any p_SUB1_resp p_SUB2_trig p_SUB2_trig_ts
where
call1_g1 : p_SUB1_resp ⊆ INT1_trig
call1_g2 : p_SUB1_resp ⊈ SUB1_resp ∪ INT1_abrt
call1_g3 : ∃ idx·idx ∈ X ∧ {idx} = p_SUB1_resp
g1 : p_SUB1_resp = {p_SUB2_trig}
call2_g1 : p_SUB2_trig ∈ X
call2_g2 : p_SUB2_trig ∉ SUB2_trig
call2_g3 : p_SUB2_trig_ts = INT1_clocks(p_SUB2_trig)
then
call1_g1 : SUB1_resp := SUB1_resp ∪ pSUB1_resp
call1_a2 : SUB1_resp_ts := SUB1_resp_ts ⩤ (pSUB1_resp × INT1_clocks[pSUB1_resp])
call2_a1 : SUB2_trig := SUB2_trig ∪ {p_SUB2_trig}
call2_a2 : SUB2_trig_ts := SUB2_trig_ts ⩤ {p_SUB2_trig ↦ p_SUB2_trig_ts}

Figure 5.22: Instantiated event e4.

In this example, the function is called once for sub-interval $SUB1$ (5.16). The call passes $SUB1$-specific parameters its name, index set X and response event $e4$.

$$TPL.SUB.RT(SUB1, X, e4) \tag{5.16}$$

We display an instantiated version of response-trigger event $e4$ (Figure 5.22). Guards and actions marked with labels $call1$ are a product of the instantiated template $TPL.ROOT.R$. Similarly, labels $call2$ mark the output of template $TPL.ROOT.T$. Parameters shown in the figure were added by the child templates $TPL.ROOT.R$ and $TPL.ROOT.T$.

Expression (5.15) sets the trigger timestamp for the succeeding sub-interval $SUB2$. Guard $g1$ ensures that the $SUB1$ response and the $SUB2$ trigger parameters match. Guard $g1$ is the only element directly generated by template $TPL.SUB.RT$ and not its child templates.

Call TPL.SUB.3 invokes template $TPL.SUB.R$ for the last sub-interval $x = TI_n$:

$$\#tc\text{: } TPL.SUB.R(x, \#\mathbf{R(x)}) \tag{TPL.SUB.3}$$

Template (Figure 5.23) guard $g1$ ensures that the response parameter carries an active but not yet responded to instance of the sub-interval n. We inject the instantiated code of this template into the interval $SUB2$ response event $e2$. The instantiated version of the template is displayed in Figure 5.24.

TEMPLATE **TPL.SUB.R(X, evts)**
@Event *ALL* ***e*** : ***evts*** ≙
X..g1 : **#parent(X)**_resp ⊆ **X**_trig

Figure 5.23: Sub-Int: event templates.

Event *e2* refines *e2* ≙
X..g1 : INT1_resp ⊆ SUB2_trig

Figure 5.24: Instantiated $TPL.SUB.R$ template.

5.1.2.1 Dynamics of SUB1 and SUB2

We explain the dynamics of sub-intervals $SUB1$ and $SUB2$ in Figure 5.25. Event $e1$ triggers the first sub-interval $SUB1$ and adds index X_1 to the sub-interval's trigger variable $INT1\_trig$ ($s2$). Note, variables used by the sub-intervals are specified in Figure 5.16. Response-trigger event $e4$ then responds to the first sub-interval and triggers $SUB2$. The index X_1 is added to $SUB1$ response and $SUB2$ trigger variables as shown in state $s3$. Finally, event $e2$ responds to the $SUB2$ interval, updating its response variable, where the latter is a reused abstract variable $INT1\_resp$.

Abort event $e3$ can occur while either $SUB1$ is active ($s2$) or $SUB2$ is active ($s3$). In such cases, index X_1 is added to the abort variable. The abort variable is common to both sub-intervals.

After responding to the sub-interval sequence ($s4$) or aborting one of the sub-intervals ($s5$ or $s6$), event $INT1\_rst$ then can clear the used-up index from the trigger, response and abort variables.

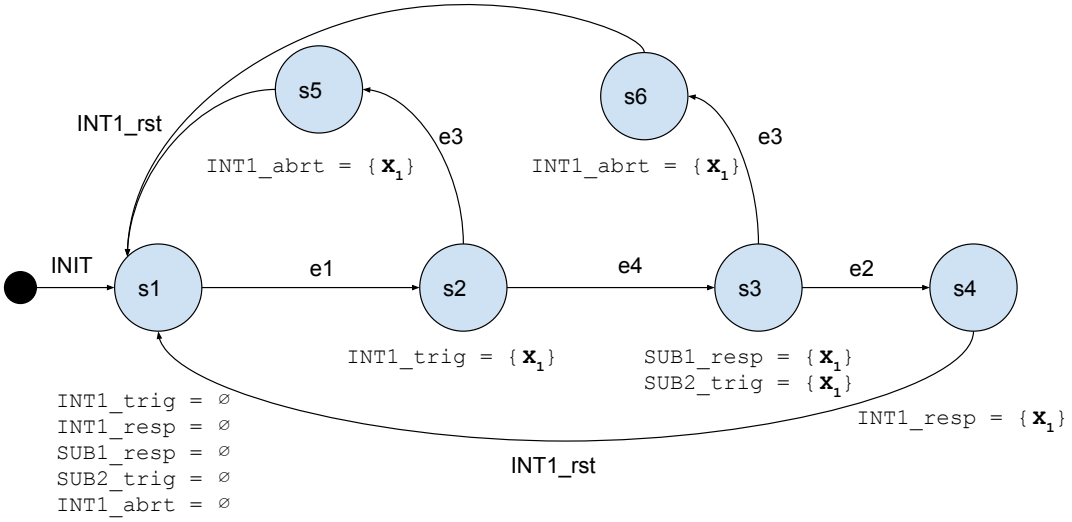

Figure 5.25: Sub-interval $SUB1$ and $SUB2$ dynamics.

5.1.2.2 Special Case: Trigger-Abort Events for Adjacent Sub-Intervals

In the previous sections, we introduced a sub-interval refinement transformation that connects adjacent sub-intervals with shared response-trigger events. The response-trigger event means that the currently active sub-interval cannot trigger the succeeding sub-interval until the timing property constraints are met. With a goal to increase the degree of modelling flexibility, we extend the transformation with a new kind of shared event – an abort-trigger event. In contrast to the response-trigger event, the new abort-trigger event is not constrained by the active sub-interval's timing properties and can trigger the succeeding sub-interval at any time. In this section, we explain how such an event works and provide the generative semantics for it.

We update the scheme in Figure 5.26 with a shared abort-trigger $e5$ that acts as an abort for sub-interval $SUB1$ and as a trigger for $SUB2$. A new abort variable $SUB1\_abrt$ is added for the sub-interval $SUB1$.

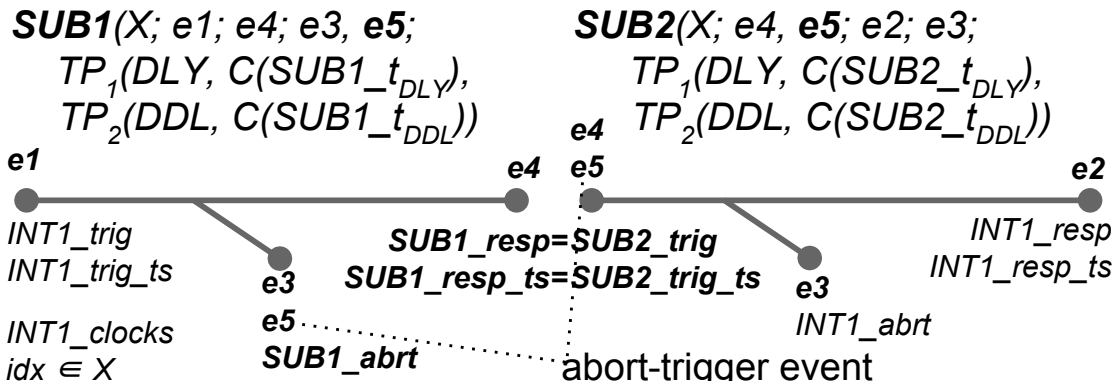

Figure 5.26: Intervals $SUB1$ and $SUB2$ with abort-trigger event

Unlike the abstract abort event $e3$ that aborts the whole sequence, event $e5$ aborts only the sub-interval and triggers the succeeding sub-interval. We distinguish abort event types by scope, relative to a timing interval. For example, interval $SUB1$ has two types of abort event. *Interval scope* abort event $e3$ affects refined $INT1$ and refining $SUB1$ timing intervals. *Local scope* abort event $e5$ aborts only the sub-interval itself. Further refinement of $SUB1$ would see both abort types as interval scope because both intervals would be affected by events $e3$ and $e5$. Local scope events must not have been used in the abstract interval's specification.

Due to the presence of the abort-trigger event $e5$, which is unconstrained by the delay timing property, sub-interval $SUB1$ can abort before reaching the delay duration $SUB1\_t_{DLY}$. In fact, the sub-interval can be aborted immediately after triggering it. This means that the delay duration of sub-interval $SUB2$ must maintain the consistency with the abstract delay duration $INT1\_t_{DLY}$. Therefore we exclude $SUB1$ from the delay duration consistency axiom (5.17). The deadline consistency axiom does not change and remains as specified in (5.12).

$$INT1\_t_{DLY} \geq SUB2\_t_{DLY} \tag{5.17}$$

Sub-intervals that are aborted by an abort-trigger event require new abort variable and consistency invariants. We invoke a new base template with call statement TPL.SUB.4. The template is called for every timing interval x that has an abort event which serves as a trigger for the succeeding sub-interval. We should remember that the previously discussed sub-interval base template (Figure 5.18) is used for all sub-intervals that do not have the abort-trigger event.

$$\#tc\text{:}\ \forall\, x \cdot x \in TI_{1..(n-1)} \wedge \mathbf{\#A(x)} \cap \mathbf{\#T(\#succ(x))} \neq \varnothing \Rightarrow TPL.SUB.Base2(x) \tag{TPL.SUB.4}$$

In our example, the template is considered only for $SUB1$ (5.18) because it is the only non-last sub-interval in the sequence. The intersection of $SUB1$ abort events $\{e3, e4\}$ and $SUB2$ trigger events $\{e4, 5\}$ is a non-empty set. Therefore template $TPL.SUB.Base2$ is called.

$$\#tc\text{: } x = SUB1 \land \{e3, e4\} \cap \{e4, e5\} \neq \varnothing \Rightarrow TPL.SUB.Base2(x) \tag{5.18}$$

We instantiate $TPL.SUB.Base2$ (Figure 5.27) for interval $SUB1$ (Figure 5.28). The generic and instantiated Event-B elements map by labels. The first three invariants $SUB1\_type\_i1 - 3$ declare $SUB1$-specific response and abort variables. Consistency invariant $SUB1\_type\_i4$ ensures that the same interval $SUB1$ instance cannot be aborted and responded to by events $e5$ and $e4$ respectively. We exclude interval scope abort indices stored in the abstract abort variable $INT1\_abrt$.

TEMPLATE TPL.SUB.Base2(X)
@INVARIANTS
X..i1 : **#resp(X)** ⊆ **#trig(X)**
X..i2 : **#resp(X)**_ts ∈ **#resp(X)** → ℕ
X..i3 : **X**_abrt ⊆ **#trig(X)**
X..i4 : (**X**_abrt \ **#parent(X)**_abrt) ∩ **#resp(X)** = ∅
X..i5 : **#trig(#succ(X))** ⊆ **#resp(X)** ∪ **X**_abrt
X..i6 : **#trig(#succ(X))**_ts ∈ **#trig(#succ(X))** → ℕ
X..i7 : **#resp(X)**_ts ⊆ **#trig(#succ(X))**_ts
X..i8 : ∀x·x ∈ **#trig(#succ(X))** ⇒ **#trig(X)**_ts(x) ≤ **#trig(#succ(X))**_ts(x)

Figure 5.27: Sub-Int: base template for intervals with the new abort-trigger events.

INVARIANTS
SUB1_type_i1 : SUB1_resp ⊆ INT1_trig
SUB1_type_i2 : SUB1_resp_ts ∈ SUB1_resp → ℕ
SUB1_type_i3 : SUB1_abrt ⊆ INT1_trig
SUB1_consist_i4 : (SUB1_abrt \ INT1_abrt) ∩ SUB1_resp = ∅
SUB1_type_i5 : SUB2_trig ⊆ SUB1_resp ∪ SUB1_abrt
SUB1_type_i6 : SUB2_trig_ts ∈ SUB2_trig → ℕ
SUB1_consist_i7 : SUB1_resp_ts ⊆ SUB2_trig_ts
SUB1_consist_i8 : ∀x·x ∈ SUB2_trig ⇒ INT1_trig_ts(x) ≤ SUB2_trig_ts(x)

Figure 5.28: Instantiated template $TPL.SUB.Base2$ for sub-interval $SUB1$.

Invariants $SUB1\_type\_i5$-6 define sub-interval $SUB2$ trigger variables. Invariant $i7$ is a consistency invariant that states a subset of $SUB2$ timestamps is equal to those of interval $SUB1$. Some of the trigger timestamps may not be present in the response timestamps in case of the abort-trigger event $e5$ occurrence – we do not record timestamps on abort. Invariant $i8$ states that all $SUB2$ trigger timestamps are greater than or equal to the corresponding $SUB1$ trigger timestamps, recorded in the reused abstract

variable $INT1\_trig\_ts$. This invariant, at first glance redundant, is required for proof obligation discharge.

We introduce a template for the new abort-trigger event type. Template call TPL.SUB.5 invokes template $TPL.SUB.A$. It passes two parameters – the list of abort-trigger events and the timing interval x that has the abort-trigger event.

$$\begin{aligned} \#tc\text{:}\ \forall x \cdot x \in TI_{1..(n-1)} \land \mathbf{\#A(x) \cap \#T(\#succ(x)) \neq \varnothing} \Rightarrow \\ TPL.SUB.AT(x, \mathbf{\#A(x) \cap \#T(\#succ(x))}) \end{aligned} \quad \text{(TPL.SUB.5)}$$

The template is nested (Figure 5.29). First, abort template $TPL.ROOT.A$ is invoked for sub-interval $\mathbf{X}(call1)$. Then, a consistency guard $g1$ is instantiated. Guard $g1$ ensures that the trigger parameter holds the same index as the abort parameter. Finally, the trigger template call ($call2$) for the succeeding sub-interval **#succ(X)** is analogous to that of the response-trigger event (Figure 5.21).

We give an example call statement for abort-trigger event $e5$, when interval x in (TPL.SUB.5) is equal to interval $SUB1$:

$$\#tc\text{:}\ \{e3, e5\} \cap \{e4, e5\} \neq \varnothing \Rightarrow TPL.SUB.AT(SUB1, \{e5\}) \quad (5.19)$$

The antecedent is true because the intersection of SUB abort event set $\{e3, e5\}$ and $SUB2$ trigger event set $\{e4, e5\}$ is non-empty. Template $TPL.SUB.AT$ is called for $SUB1$ passing one abort-trigger event $e5$ in the list. Figure 5.30 displays the instantiated version of template $TPL.SUB.AT$ for abort-trigger event $e5$. Label $call1$ and $call2$ mark Event-B output for abort $TPL.ROOT.A$ and trigger $TPL.ROOT.T$ templates respectively.

We update $SUB1$ abort variable $SUB1\_abrt$ (5.20) when abort event $e3$ fires. The update is required because the new abort variable registers all $SUB1$ abort event occurrences – local ($e5$) and interval ($e3$) scope.

$$SUB1\_abrt := SUB1\_abrt \cup p\_INT1\_abrt \quad (5.20)$$

Sub-intervals with the abort-trigger event require a more complex deadline template. Call statement TPL.SUB.6 invokes a special-case deadline timing property template $TPL.SUB.DDL\_AT$. The call passes three parameters to the template: the timing

TEMPLATE **TPL.SUB.AT(X, IDX, evts, ts)**
@Event *ALL* $e : evts \widehat{=}$
@where
call1 : $\#tc:$ TPL.ROOT.A(X, e)
X..g1 : {p_#trig(#succ(X))} = p_X_abrt
call2 : $\#tc:$ TPL.ROOT.T(#succ(X), IDX, e, #parent(X)_clocks(p_#trig(X))

Figure 5.29: Sub-Int: abort-trigger event template.

Event $e5 \widehat{=}$
any p_SUB1_abrt p_SUB2_trig p_SUB2_trig_ts
where
call1_g1 : p_SUB1_abrt $\subseteq$ INT1_trig $\setminus$ (SUB1_resp $\cup$ SUB1_abrt)
call1_g2 : INT1_trig $\setminus$ (SUB1_resp $\cup$ SUB1_abrt) $\neq \varnothing \Rightarrow$
$(\exists$ idx $\cdot$ idx $\in$ INT1_trig $\setminus$ (SUB1_resp $\cup$ SUB1_abrt) $\wedge$ p_SUB1_abrt = {idx})

SUB1_consist_g1 : {p_SUB2_trig} = p_SUB1_abrt
call2_g1 : p_SUB2_trig $\in$ X
call2_g2 : p_SUB2_trig $\notin$ SUB2_trig
call2_g3 : p_SUB2_trig_ts = INT1_clocks(p_SUB2_trig)
then
call1_a1 : SUB1_abrt := SUB1_abrt $\cup$ p_SUB1_abrt
call2_a1 : SUB2_trig := SUB2_trig $\cup$ {p_SUB2_trig}
call2_a2 : SUB2_trig_ts := SUB2_trig_ts $\leftarrowtail$ {p_SUB2_trig $\mapsto$ p_SUB2_trig_ts}

Figure 5.30: Instantiated abort-trigger event $e5$.

interval x, its abort-trigger event list $\#\mathbf{A(x)} \cap \#\mathbf{T(\#succ(x))}$ and its response event list $\#\mathbf{R(x)}$.

$$\begin{aligned} &\#tc: \forall\, x \cdot x \in TI_{1..(n-1)} \wedge \#\mathbf{A(x)} \cap \#\mathbf{T(\#succ(x))} \neq \varnothing \Rightarrow \\ &TPL.SUB.DDL\_AT(x, \#\mathbf{A(x)} \cap \#\mathbf{T(\#succ(x))}, \#\mathbf{R(x)}) \end{aligned} \tag{TPL.SUB.6}$$

Firstly, the template (Figure 5.31) invokes abstract deadline timing property template ($call1$). It then instantiates extra invariants and guards, which help to discharge the proof obligations. The guards are added to all abort-trigger and response-trigger events in variables *atEvts* and *rEvts* respectively. We explain the invariants and guards in the instantiated version of the template in further paragraphs.

We give a concrete version of the template call for sub-interval $SUB1$, in which the first parameter is the interval name, the second parameter is the abort-trigger event $e5$ and the third parameter is the response-trigger event $e4$.

$$TPL.SUB.DDL\_AT(SUB1, \{e5\}, \{e4\}) \tag{5.21}$$

TEMPLATE **TPL.SUB.AT.DDL_AT(X, atEvts, rEvts)**

@INVARIANTS
call1 : *#tc:* TPL.ROOT.DDL(x)
X..i1 : **X_abrt** \ **#parent(X)_abrt** ⊆ **#trig(#succ(X))** \ **#parent(X)_abrt**
X..i2 : ∀x·x ∈ **#trig(#succ(X))** ⇒ **#trig(#succ(X))_ts(x)** ≤ **X_t**$_{DDL}$

@Event *ALL e* : ***atEvts*** ≙
@where
X..at_g1 : ∀idx·idx ∈ p_**X_abrt** ⇒ **#parent(X)_clocks(idx)** ≤ **X_t**$_{DDL}$

@Event *ALL e* : ***rEvts*** ≙
@where
X..r_g1 : ∀idx·idx ∈ p_**#resp(X)** ⇒ **#parent(X)_clocks(x)** ≤ **X_t**$_{DDL}$

Figure 5.31: Sub-Int: deadline template with additional invariants and guards.

INVARIANTS
call1 : ∀idx·idx ∈ INT1_trig ∧ idx ∉ SUB1_resp ∪ SUB1_abrt⇒
INT1_clocks(idx) ≤ INT1_trig_ts(idx) + SUB1_t$_{DDL}$
call1 : ∀idx·idx ∈ INT1_trig ∧ idx ∈ SUB1_resp ⇒ SUB1_resp_ts(idx) ≤ INT1_trig_ts(idx) + SUB1_t$_{DDL}$
SUB1_i1 : SUB1_abrt \ INT1_abrt ⊆ SUB2_trig \ INT1_abrt
SUB1_i2 : ∀x·x ∈ SUB2_trig ⇒ SUB2_trig_ts(x) ≤ SUB1_t$_{DDL}$

Event *e*5 ≙
where
SUB1_at_g1 : ∀idx·idx ∈ p_SUB1_abrt ⇒ INT1_clocks(idx) ≤ SUB1_t$_{DDL}$

Event *e*4 ≙
where
SUB1_r_g1 : ∀idx·idx ∈ p_SUB1_resp ⇒ INT1_clocks(x) ≤ SUB1_t$_{DDL}$

Event *Tick* ≙
where
call1 : ∀idx·idx ∈ INT1_trig ∧ idx ∉ SUB1_resp ∪ SUB1_abrt⇒
INT1_clocks(idx) + 1 ≤ INT1_trig_ts(idx) + SUB1_t$_{DDL}$

Figure 5.32: Instantiated deadline template *SUB.AT.DDL_AT* for *SUB*1.

The instantiated version of the *SUB.AT.DDL_AT* template is displayed in Figure 5.32. Invariants and guards labelled *call*1 mark elements produced by the child template *TPL.ROOT.DDL*. Invariant *i*1 ensures that all indices, which are contained by *SUB*1 abort variable *SUB*1_*abrt*, are present in the sub-interval *SUB*2 trigger variable *SUB*2_*trig*. We exclude indices that are present in abstract abort variable *INT*1_*abrt*. Invariant *i*2 ensures that all succeeding sub-interval are triggered within the preceding sub-interval's deadline duration. Abort-trigger event *e*5 and response-trigger event *e*4 are added a guard that respects consistency invariant *i*2. The guard says that the shared events can occur only within *SUB*1 deadline duration.

The delay timing property does not require additional invariants and is identical to that of the alternative transformation (TPL.ALT.5).

5.1.2.3 Dynamics of SUB1 with Abort-Trigger Event and SUB2

We show the dynamics of the sub-interval sequence, when sub-interval $SUB1$ has a special case abort-trigger variable $e5$ (Figure 5.33). We remember that variables used by $SUB1$ and $SUB2$ are defined in the updated schematics in Figure 5.26.

In contrast to the previously described sub-interval dynamics graph in Figure 5.25, the new graph (Figure 5.33) includes the new abort variable $SUB1\_abrt$ for $SUB1$ (highlighted in bold). The dynamics of the sub-interval sequence differ in the old state $s5$ and new state $s7$.

In the case where the active $SUB1$ interval instance is aborted ($s2$) by interval scope abort event $e3$, the new $SUB1$-specific abort variable is updated as well (state $s5$). The new state $s7$ shows the state-change when the abort-trigger event $e5$ fires. The interval $SUB1$ is aborted by adding index X_1 to the abort variable $SUB1\_abrt$ and sub-interval $SUB2$ trigger variable $SUB2\_trig$. Their indices are synchronised.

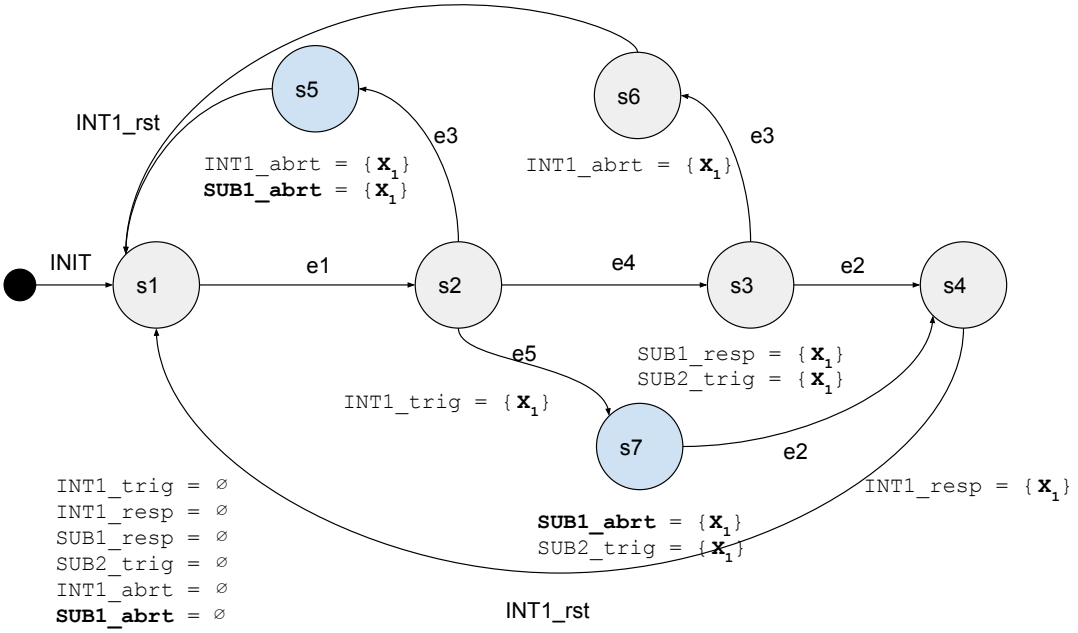

Figure 5.33: Special case: Sub-interval interval $SUB1$ and $SUB2$ dynamics with abort-response event $e5$.

5.1.3 Abort-To-Response Transformation

In this section, we introduce an Abort-to-Response refinement transformation. The transformation transforms the abstract interval's abort variables to response. For instance, an abort at abstract interval level may be refined to a sub-interval sequence, where the abort is only permitted in one sub-interval or the other.

We explain the motivation for such a transformation with the example in Figure 5.34. Consider interval $INT1$ in the abstract level $m0$. The requirements for $INT1$ define event $e3$ as an abort event without timing restrictions on its occurrence. The interval is then further transformed into a sequence of sub-intervals $SUB1$ and $SUB2$ to reflect additional requirements. At this stage, we learn from the requirements at the concrete level that event $e3$ for sub-interval $SUB1$ must be restricted in a similar way to a response-trigger event $e4$. With the help of the Abort-to-Response refinement transformation, we can fulfil this requirement in a methodical way. The Abort-to-Response transformation is compositional, the modified $SUB2$ sub-interval can be further refined using other refinement transformations.

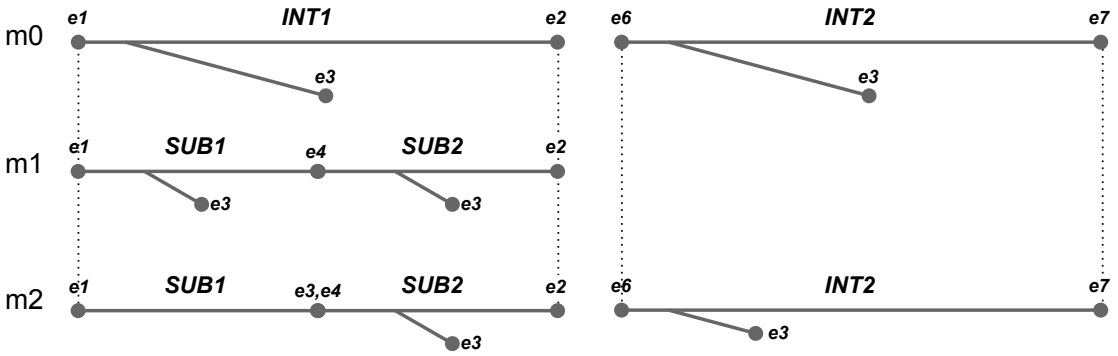

Figure 5.34: Abort-to-Response transformation example.

The Abort-to-Response transformation requires an interval with at least one abort event. It then transforms chosen abort events to response events. If the transformed abort event serves as an abort for other intervals, it retains that role for them. In the given example (Figure 5.34) refinement $m3$, both intervals $SUB1$ and $INT2$ start at the same time triggered by event $e1$. Whenever $SUB1$ is responded to by the transformed event $e3$, it aborts the $INT2$ instance. The occurrence of $e3$ is not constrained in $INT2$. The refinement transformation provides the timing interval transformation without breaking the timing interval structure and maintaining the compositionality.

Here we give a concrete transformation example; generic Abort-to-Response templates are in section A.1. We use Abort-to-Response refinement transformation to refine abort event $e3$ of interval $INT1$ - which can fire at any time - into the response $e3\_atr$. In this example, abort variable $INT1\_abrt$ is no longer used for abort purposes by the ATR interval because there are no abort events left. However, the variable is retained for the consistency of further refinements. The interval reuses abstract trigger variables (Figure 5.35).

We start the transformation by introducing new response variables that treat abort indices of interval ATR as the response. The new response variable ATR_resp (5.22) is defined as the abstract interval $INT1$ response indices, together with the $INT1$ abort indices (5.22). The latter are precisely those indices that are triggered and aborted. Variable $INT1\_abrt$ holds abort indices for all intervals in the refinement chain, and

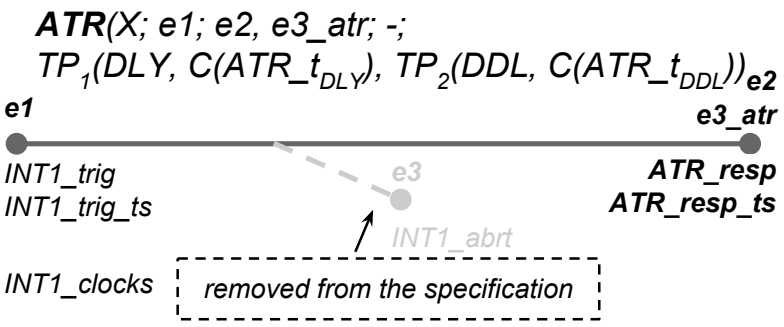

Figure 5.35: Interval *ATR*

expression ($INT1\_trig \cap INT1\_abrt$) filters out only those indices that are associated with interval *ATR*. We remind the reader that, according to Figure 5.35, variable $INT1\_trig$ holds interval's trigger events. Therefore the intersection of $INT1\_trig$ and $INT1\_abrt$ returns only those indices that represent aborted interval *ATR* instances. We then introduce a new timestamp (5.23), which is associated with the new response variable.

$$\mathbf{ATR\_resp} = INT1\_resp \cup (INT1\_trig \cap INT1\_abrt) \tag{5.22}$$

$$\mathbf{ATR\_resp\_ts} \in \mathbf{ATR\_resp} \rightarrow \mathbb{N} \tag{5.23}$$

The transformed response event $e3\_atr$ (Figure 5.36) is injected with the guard ATR_resp_g1. The guard restricts the abstract abort parameter p_INT1_abrt to active but not yet responded to indices of *ATR*. We highlight the new response variable in bold. Due to this guard, event $e3\_atr$ can only be executed when *ATR* is active and satisfies its timing property restrictions. Upon execution of event $e3\_atr$, the index is recorded to the new response variable (ATR_resp_a1) and the timestamp is taken (ATR_resp_a2).

INVARIANTS

ATR_dly_i1 : $\forall idx \cdot idx \in \text{INT1\_trig} \land idx \in \mathbf{ATR\_resp} \Rightarrow \mathbf{ATR\_resp\_ts}(idx) \geq \mathbf{ATR\_t_{DLY}}$

Event $e3\_atr$ refines $e3 \mathrel{\widehat{=}}$
where
Grds
p_ATR_consist_grd : $\text{p\_ATR\_resp} = \text{p\_INT1\_abrt}$
ATR_resp_g1 : $\text{p\_ATR\_resp} \subseteq \text{INT1\_trig}$
ATR_resp_g2 : $\text{p\_ATR\_resp} \cap (\mathbf{ATR\_resp} \cup \text{INT1\_abrt}) = \varnothing$
ATR_dly_g1 : $\forall idx \cdot idx \in \text{p\_INT1\_abrt} \Rightarrow \text{INT1\_clocks}(idx) \geq \text{INT1\_trig\_ts}(idx) + \mathbf{ATR\_t_{DLY}}$
then
Acts
ATR_resp_a1 : $\mathbf{ATR\_resp} := \mathbf{ATR\_resp} \cup \text{p\_INT1\_abrt}$
ATR_resp_a2 : $\mathbf{ATR\_resp\_ts} := \mathbf{ATR\_resp\_ts} \leftarrow (\text{p\_INT1\_abrt} \times \text{INT1\_clocks}[\text{p\_INT1\_abrt}])$
end

Figure 5.36: Evt. $e3\_atr$: injected Abort-to-Response code.

As before for $ALT1$ in subsection 5.1.1, we generate the delay timing property by invoking template call analogous to (TPL.ALT.4). Invariant ATR_dly_i1 is instantiated over new and reused variables as displayed in (Figure 5.36). To preserve this invariant, guard ATR_dly_g1 is injected into $e3\_atr$, the response event of ATR. The Event-B generation for the deadline timing property is achieved by executing call statement, analogous to (TPL.ALT.4).

Axioms (5.24) and (5.25) ensure consistency between the $INT1$ and ATR interval.

$$\mathbf{ATR\_t_{DLY}} \geq INT1\_t_{DLY} \qquad (5.24) \qquad\qquad \mathbf{ATR\_t_{DDL}} \leq INT1\_t_{DDL} \qquad (5.25)$$

Finally, interval ATR can be further elaborated with the refinement transformations. However, further refined intervals cannot abort.

5.1.3.1 Dynamics of the ATR Interval

We demonstrate the dynamics of the example model with a graph in Figure 5.37. We trigger interval ATR by firing event $e1$. The interval's trigger variable gets updated with the instance index X_1 ($s2$). The interval can then be responded to with the response event $e2$. The interval's response variable is then added to index X_1 ($s3$). Note that the abstract response variable, which is no longer used as a response variable for interval ATR, is updated as well. Similarly, if the active interval ATR instance is responded to by the transformed event $e3\_atr$, interval index X_1 is added to the new response variable ATR_resp and the abstract abort variable $INT1\_abrt$ ($s5$).

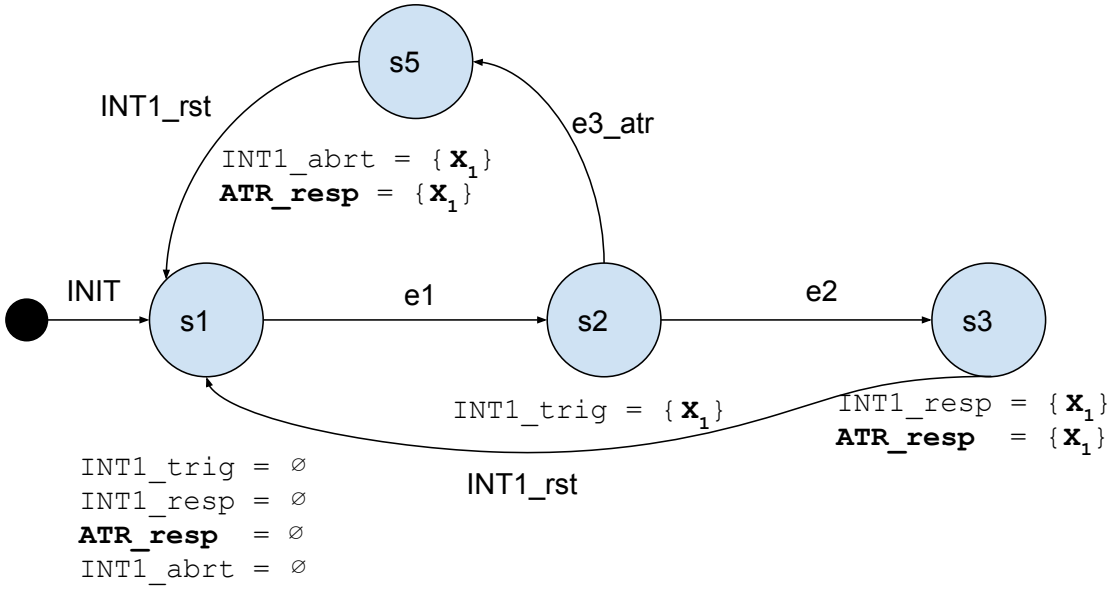

Figure 5.37: ATR $INT1\_atr$ dynamics.

5.1.4 Single-to-Multi Transformation

We introduce a Single-to-Multi refinement transformation. The transformation refines an abstract interval to a *Single-to-Multi interval*. The Single-to-Multi interval elaborates each refined abstract interval's instance with a predefined set of *sub-instances*. A set of such sub-instances is called a *group*.

In typical situations, set X is sufficient to index timing interval instances. Let's call such a set a single-dimension Cartesian product. Single-to-Multi and Retry (discussed in subsection 5.1.5) refinement transformations use an n-fold Cartesian product to identify sub-instances for each abstract interval's instance. The n-fold Cartesian product is always one dimension greater than the abstract index set, e.g. $X \times Y$, where X is the abstract interval's index set.

All sub-instances in a group are recorded in the same set of variables; sub-instances always have identical timing property constraints, and further refinement thereof affects all sub-instances. In contrast, sub-intervals (subsection 5.1.2) are treated as distinct intervals with their variables and can be further refined individually.

For example purposes, we refine $INT1$ to an interval STM by applying the Single-to-Multi transformation. We explain the requirements for Single-to-Multi refinement transformation along the narration of this example. The index set of the STM interval is defined as $X \times Y$. Where $card(Y) = 4$, meaning that each interval $INT1$ instance is represented by a group of four Single-to-Multi sub-instances. Interval STM sub-instance delay (5.26) and deadline (5.27) durations must be consistent with the corresponding abstract interval $INT1$ durations.

$$\mathbf{card(Y)} * \mathbf{STM\_t_{DLY}} \geq INT1\_t_{DLY} \tag{5.26}$$

$$\mathbf{card(Y)} * \mathbf{STM\_t_{DDL}} \leq INT1\_t_{DDL} \tag{5.27}$$

Consider the example scenario in Figure 5.38 that illustrates the behaviour of the STM interval. The first STM sub-instance can only be triggered by a refined trigger event – in this case event $e1$. Subsequent STM sub-instances can only be triggered by newly introduced trigger events. In our example, new event $e1\_2$ triggers sub-instances 2, 3 and 4. All but the last STM sub-instance can only be responded to by a new response event – in this example, event $e2\_2$. The group completes when the last sub-instance is responded to by the refined response event $e2$. At any point in time, abort event $e3$ can abort the whole STM group. While the SMT group is active, at least one sub-instance must be running. Sub-instances are concurrent.

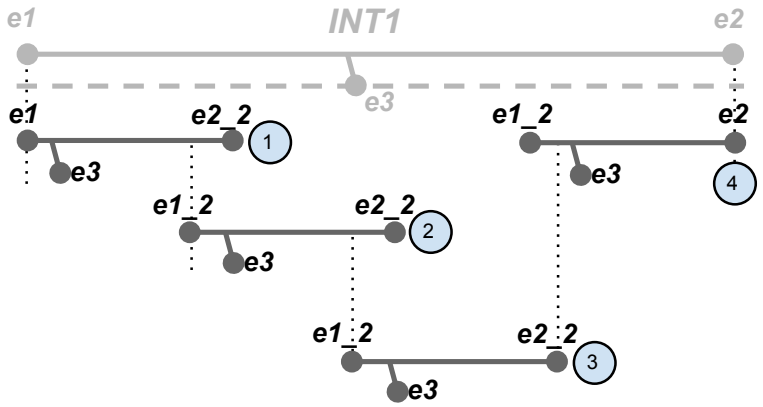

Figure 5.38: Interval *STM* example scenario

The *STM* specification (5.28) reflects the addition of one new trigger event $e1\_2$, one new response event $e2\_2$ and a new instance index set.

$$\begin{aligned} &STM(X \times Y, e1, e1\_2;\ e2, e2\_2;\ e3; \\ &\quad TP_1(DLY, C(STM\_t_{DLY})), TP_2(DDL, C(STM\_t_{DDL}))) \end{aligned} \tag{5.28}$$

The interval, as displayed in Figure 5.39, uses new variables and one auxiliary variable STM_trig_intr. The semantics of Single-to-Multi refinement transformation is achieved by extending already available templates. In this section, we cover only the instantiated Event-B.

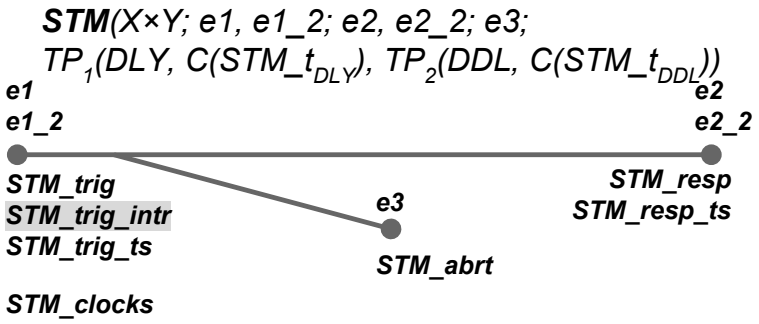

Figure 5.39: Interval *STM*

The general principle on how interval *STM* operates is analogous to the previously discussed. We point out the reusability aspect and provide generic templates and the corresponding template calls for Single-to-Multi refinement transformation in section A.2. In this section, we discuss only the behaviour that is specific to the Single-to-Multi refinement transformation. We do this on the instantiated interval *STM* example.

We start off by constructing interval *STM* variables and consistency invariants (Figure 5.40). Element *Invs* denotes other invariants, such as those related to abstract interval $INT1$. Marked with label *call*1 is the reused semantics from template $TPL.ROOT.Base$

(Figure 4.14). A new auxiliary trigger variable *STM_trig_intr* is declared. The variable holds instance indices triggered by newly introduced trigger events – in this case event *e*1_2.

INVARIANTS

Invs

call1 : STM_trig ⊆ X × Y

call1 : STM_trig_ts ∈ STM_trig → ℕ

call1 : STM_resp ⊆ STM_trig

call1 : STM_resp_ts ∈ STM_resp → ℕ

call1 : STM_abrt ⊆ STM_trig

call1 : STM_clocks ∈ STM_trig → ℕ

call1 : STM_resp ∩ STM_abrt = ∅

STM_type_i1 : STM_trig_intr ⊆ STM_trig

STM_consist_i1 : dom(STM_trig) = INT1_trig

STM_consist_i2 : dom(STM_abrt) = INT1_abrt

STM_consist_i3 : INT1_resp = {x | STM_resp[{x}] = Y}

STM_consist_i4 : ∀x·x ∈ dom(STM_trig\(STM_resp∪STM_abrt))⇔x ∈ INT1_trig\(INT1_resp∪INT1_abrt)

STM_consist_i5 : ∀x, y·x ↦ y ∈ STM_trig\(STM_resp∪STM_abrt)⇒STM_clocks(x ↦ y) = INT1_clocks(x)

STM_consist_i6 : ∀x, y·x ↦ y ∈ dom(STM_trig_intr ⩤ STM_trig_ts) ⇒ STM_trig_ts(x ↦ y) = INT1_trig_ts(x)

Figure 5.40: STM gluing invariants.

The six gluing invariants *STM_consist_i*1-6, displayed in Figure 5.40, express the following requirements:

i1 – if the abstract interval instance has been triggered, at least one sub-instance of the corresponding *STM* sub-instance group is active.

i2 – the aborted abstract interval instance corresponds to at least one aborted *STM* sub-instance.

i3 – responded to *INT*1 instances and their corresponding *STM* groups match.

i4 – for every running abstract interval's instance there is at least one active *STM* sub-instance running.

i5 – every abstract interval instance's clock matches the corresponding groups sub-instance clocks. This invariant also ensures that all group sub-instance clocks are synced.

i6 – all *STM* group start timestamps match the abstract interval trigger timestamps. Note that we use the auxiliary variable to filter out timestamps that were recorded by newly introduced trigger events.

Events for Single-to-Multi refinement transformation inherit the behaviour from reused templates, as discussed in section 4.5. In the following paragraphs, we briefly remind

the reader about the general behaviour for each *STM* event and give detail only on the new elements with Single-to-Multi -specific behaviour.

Upon firing trigger events $e1$ and $e1\_2$, the new sub-instance index is added to the *STM* trigger variables. The additional guard STM_trig_g1 in the refined event $e1$ (Figure 5.42) synchronises the abstract p_INT1_trig and concrete p_STM_trig trigger parameters. Similarly, guard STM_trig_grd1 in the newly introduced event $e1\_2$ (Figure 5.43) ensures that the sub-instance to be triggered belongs to the already active *STM* group. Elements *Grds* and *Acts* denote other guards and actions, including ones related to the abstract interval $INT1$.

Single-to-Multi response events respond to an already existing sub-instance in the sequence by adding its index to the *STM* response variables. Guard STM_resp_g1 in the new response event event $e2\_2$ (Figure 5.44) ensures that the event responds to the non-last sub-instance in the group. The last sub-instance in the group can only be responded to by the refined response event – in this case event $e2$. Two guards ensure the consistent behaviour of the refined response events: guard STM_resp_grd1 synchronises abstract p_INT1_resp and concrete p_STM_resp response parameters; guard STM_resp_grd2 ensures that the event responds to the last sub-instance in the group.

In contrast to the abort event for the *ROOT* timing interval (Figure 4.25) – which can abort one interval instance at a time – the abort event $e3$ (Figure 5.41) for the *STM* interval can abort a set of sub-instances. This change in behaviour comes from the requirement expressed in invariant $STM\_consist\_i2$ (Figure 5.40). The invariant states that if the abstract interval $INT1$ instance is aborted, then all of the *STM* sub-instances, corresponding to the abstract instance, must be aborted as well. We remind the readers that *STM* can have multiple group-related sub-instances running in parallel. We ensure this behaviour in the abort event $e3$ with guard STM_abrt_grd1 (Figure 5.41). The guard ensures that upon event execution all active group-related *STM* sub-instances corresponding to abstract interval $INT1$ abort parameter p_INT1_abrt will be aborted.

$$\texttt{STM\_abrt\_g1} : \texttt{p\_STM\_abrt} = \texttt{p\_INT1\_abrt} \lhd (\texttt{STM\_trig} \setminus (\texttt{STM\_resp} \cup \texttt{STM\_abrt}))$$

Figure 5.41: Event $e3$ extra guards.

The delay and deadline timing property invariants are identical to those defined in Figure 4.16 and Figure 4.17 respectively. The proof of axiom (5.26) is facilitated by two additional delay timing property invariants STM_dly_i1 and STM_dly_i2 (Figure 5.46). Invariant STM_dly_i1 states that any two timestamps of sub-instances of the same group must be separated by at least STM_t_{DLY} time units. The second invariant STM_dly_i2 states that if all but the last of the sub-instances were responded to, then the last active sub-instance's delay threshold must be greater than or equal to

that of the abstract interval $INT1$. The invariants complement the delay duration consistency axiom between the concrete STM and abstract $INT1$ invariants (5.26). Guards marked STM_dly_g* in events $e1$, $e2\_2$ and $e2$ maintain the invariants.

An additional deadline invariant STM_ddl_i1 (Figure 5.46) ensures that none of the triggered STM interval sub-instances can finish beyond the abstract interval's deadline. A guard STM_ddl_g1 in the new event $e1\_2$ then maintains the invariant.

Interval STM reuses abstract interval $INT1$'s reset event $INT1\_rst$. We add a new parameter p_STM_reset, defined by guard $STM\_rst\_grd\_1$. The parameter holds all sub-instances relevant to the abstract indices held in the abstract interval $INT1$ reset parameter p_STM_reset. For example, if the abstract interval $INT1$ has an instance with index X_1 that has been responded to, then parameter p_STM_reset will reset all $X_1 \times Y$ indices. The reset actions $STMrst\_act\_1$-6 are analogous to the $TPL.ROOT.RST$ template Figure 4.15.

5.1.4.1 Dynamics of STM interval

We outline the dynamics of the SMT interval in Figure 5.38. The graph reflects the scenario depicted in Figure 5.38. The first sub-instance in the group is triggered by firing event $e1$. An index is added to abstract and concrete variables ($s2$). Note that the domain of the sub-instance index $X_1 \mapsto Y_1$ matches the abstract index X_1. A second sub-instance is triggered by event $e1\_2$ and the system enters state $s3$. In this state, a new index $X_1 \mapsto Y_2$ is added to the STM trigger variable STM_trig and the auxiliary variable STM_trig_intr. We remind the reader that the latter records indices triggered by the newly introduced trigger event $e1\_2$. Then, sub-instance $X_1 \mapsto Y_1$ responded to by the intermediate event $e2\_2$. The index is recorded to the response variable ($s4$). We repeat the trigger and response actions twice, adding two more sub-instances $X_1 \mapsto Y_3$ and $X_1 \mapsto Y_4$ and responding to $X_1 \mapsto Y_2$ and $X_1 \mapsto Y_3$. Finally, refined event $e2$ responds to the last sub-instance $X_1 \mapsto Y_4$ in group. The abstract interval is responded to as well.

In case the abort event $e3$ occurs while in state $s3$, the index is added to abstract abort variable and all active sub-instance indices added to the concrete variable. The abort event can occur at any point and state $s3$ is given just as an example.

5.1.5 Retry Transformation

The Retry refinement transformation transforms an abstract interval instance to a number of sub-instances, or attempts, collectively called a sequence. At any point in time,

Event *e1* refines *e1* $\widehat{=}$

any p_INT1_trig p_INT1_trig_ts p_STM_trig p_STM_trig_ts
where
Grds
STM_trig_g1 : $p\_INT1\_trig \in dom(\{p\_STM\_trig\})$
STM_dly_g1 : $\forall x, y \cdot x \mapsto y \in dom(\{p\_STM\_trig\}) \lhd STM\_trig \Rightarrow STM\_trig\_ts(x \mapsto y) + STM\_t_{DLY} \leq p\_STM\_trig\_ts$
STM_dly_g2 : $STM\_resp[dom(\{p\_STM\_trig\})] \neq Y \setminus ran(\{p\_STM\_trig\})$
...

Figure 5.42: Event $e1$

Event *e1_2* $\widehat{=}$

any p_STM_trig p_STM_trig_ts
where
Grds
STM_trig_g1 : $dom(\{p\_STM\_trig\}) \subseteq dom(STM\_trig \setminus (STM\_resp \cup STM\_abrt))$
STM_dly_g1 : $\forall x, y \cdot x \mapsto y \in dom(\{p\_STM\_trig\}) \lhd STM\_trig \Rightarrow STM\_trig\_ts(x \mapsto y) + STM\_t_{DLY} \leq p\_STM\_trig\_ts$
STM_ddl_g1 : $\forall x \cdot x \in dom(STM\_trig\_intr \lhd\!\!\!- STM\_clocks) \Rightarrow STM\_clocks(x) - STM\_trig\_ts(x) + STM\_t_{DDL} \leq INT1\_t_{DDL}$
...

Figure 5.43: Event $e1\_2$

Event *e2_2* $\widehat{=}$

any p_STM_resp
where
Grds
STM_resp_g1 : $(dom(\{p\_STM\_resp\}) \lhd STM\_trig) \setminus (STM\_resp \cup STM\_abrt \cup \{p\_STM\_resp\}) \neq \varnothing$
STM_dly_g1 : $p\_STM\_dly\_ts\_aux \in (STM\_trig\_intr \lhd\!\!\!- STM\_trig\_ts)[dom(\{p\_STM\_resp\}) \lhd (STM\_trig)]$
STM_dly_g2 : $\forall x, y \cdot x \mapsto y \in dom(\{p\_STM\_resp\}) \lhd dom(STM\_trig\_ts) \land (STM\_resp \cup \{p\_STM\_resp\})[\{x\}] = Y \setminus \{y\} \land$
$x \mapsto y \notin STM\_resp \cup STM\_abrt \Rightarrow STM\_trig\_ts(x \mapsto y) + STM\_t_{DLY} \geq p\_STM\_dly\_ts\_aux + INT1\_t_{DLY}$
...

Figure 5.44: Event $e2\_2$

Event *e2* refines *e2* $\widehat{=}$

any p_INT1_resp p_STM_resp
where
Grds
STM_resp_g1 : $p\_INT1\_resp \in dom(\{p\_STM\_resp\})$
STM_resp_g2 : $(dom(\{p\_STM\_resp\}) \times Y) \subseteq STM\_resp \cup \{p\_STM\_resp\}$
STM_dly_g1 : $STM\_resp[dom(\{p\_STM\_resp\})] = Y \setminus ran(\{p\_STM\_resp\})$
...

Figure 5.45: Event $e2$

INVARIANTS
Invs
STM_dly_i1 : $\forall x, y, z \cdot x \mapsto y \in dom(STM\_trig\_ts) \land x \mapsto z \in dom(STM\_trig\_ts) \land y \neq z \Rightarrow$
$(STM\_trig\_ts(x \mapsto y) + STM\_t_{DLY} \leq STM\_trig\_ts(x \mapsto z)) \lor$
$(STM\_trig\_ts(x \mapsto z) + STM\_t_{DLY} \leq STM\_trig\_ts(x \mapsto y))$
STM_dly_i2 : $\forall x, y \cdot x \mapsto y \in dom(STM\_trig\_ts) \land STM\_resp[\{x\}] = Y \setminus \{y\} \land$
$x \mapsto y \notin STM\_resp \cup STM\_abrt \Rightarrow STM\_trig\_ts(x \mapsto y) + STM\_t_{DLY} \geq INT1\_trig\_ts(x) + INT1\_t_{DLY}$
STM_dly_i1 : $\forall idx, x \cdot idx \mapsto x \in STM\_trig \land idx \in INT1\_trig \Rightarrow$
$INT1\_trig\_ts(idx) + INT1\_t_{DDL} \geq STM\_trig\_ts(idx \mapsto x) + STM\_t_{DDL}$

Figure 5.46: STM timing property invariants

Event *INT1_rst* refines *INT1_rst* $\widehat{=}$

any p_INT1_reset p_STM_reset
where
Grds
STMrst_grd_1 : p_STM_reset = p_INT1_reset × Y
then
Acts
STMrst_act_1 : STM_trig := STM_trig \ p_STM_reset
STMrst_act_2 : STM_trig_ts := p_STM_reset ⩤ STM_trig_ts
STMrst_act_3 : STM_clocks := p_STM_reset ⩤ STM_clocks
STMrst_act_4 : STM_resp := STM_resp \ p_STM_reset
STMrst_act_5 : STM_resp_ts := p_STM_reset ⩤ STM_resp_ts
STMrst_act_6 : STM_abrt := STM_abrt \ p_STM_reset
STMrst_act_7 : STM_trig_intr := STM_trig_intr \ p_STM_reset
end

Figure 5.47: Elaborated reset event *INT*1_*rst*.

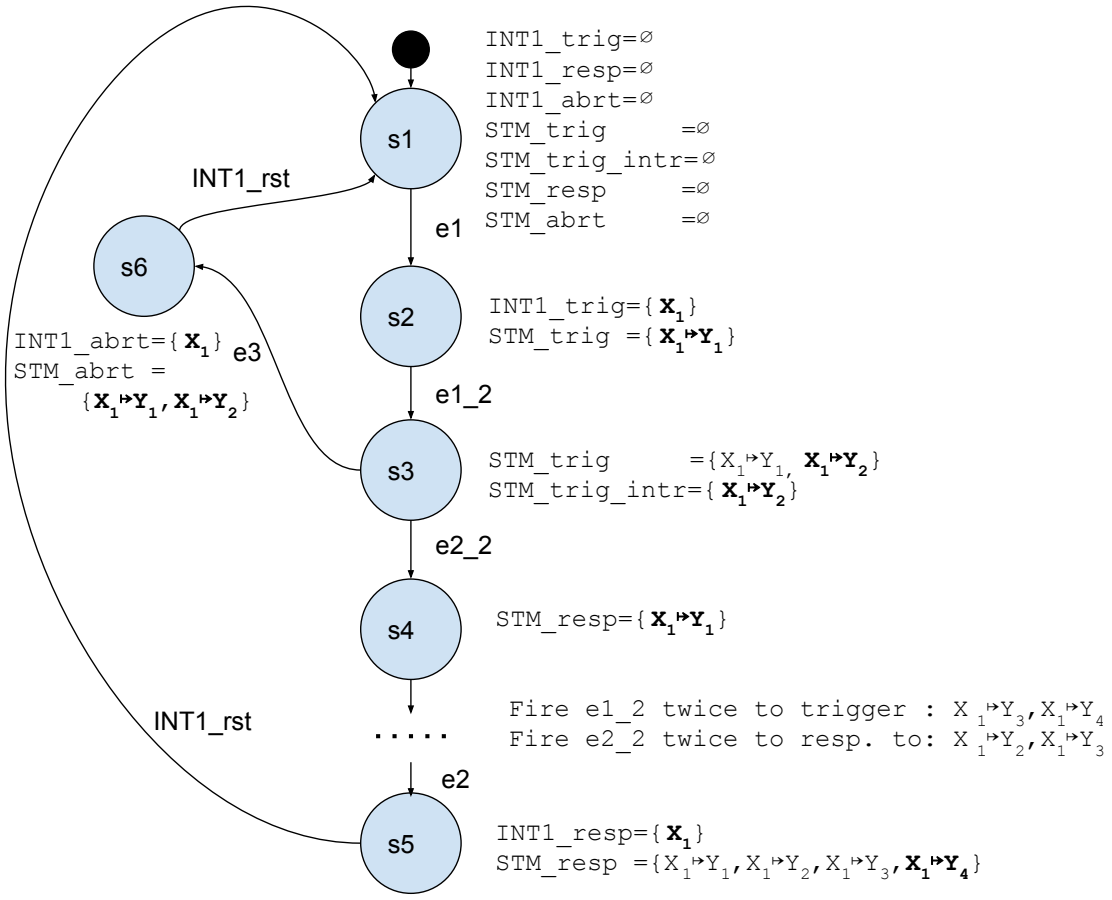

Figure 5.48: Single-to-Multi interval *STM* dynamics.

only one attempt can be active. In case it fails, another attempt is triggered. The sequence completes after the first successfully finished attempt. Otherwise, it can run until it reaches the maximum allowed attempt limit and is blocked. The blocked sequence then must be aborted with the abort event.

We explain the transformation through an example where we refine *INT*1 into interval *RTR*. Interval *RTR* elaborates *INT*1 instance to four *RTR* sub-instances, or attempts.

The product of all Retry sub-instance deadline durations must be less or equal to the abstract deadline duration (5.29), where $card(Y) = 4$. The delay duration of a single Retry attempt must be greater or equal to the abstract counterpart (5.30).

$$\mathbf{card(Y)} * \mathbf{RTR\_t_{DDL}} \leq INT1\_t_{DDL} \tag{5.29}$$

$$\mathbf{RTR\_t_{DLY}} \geq INT1\_t_{DLY} \tag{5.30}$$

An example sequence execution scenario is displayed in Figure 5.49. Only refined trigger events can initiate the sequence. Thus, the first *RTR* attempt is triggered by the refined event $e1$. Attempt failure is modelled with a newly introduced abort-trigger event and can occur at any point in time. The abort-trigger event aborts the currently active sub-instance and triggers a new one. In this example, we model the failure of attempts (1) and (2) with the abort-trigger event e_r. The sequence finishes when the third attempt is responded to by event $e2$. Only refined interval response events can respond to the sequence. Alternatively, if event $e2$ did not occur, the sequence would fail again and reach the limit of instances (4). The sequence would then either be responded to or aborted. Refined abort events, such as $e3$, can abort the sequence at any point in time.

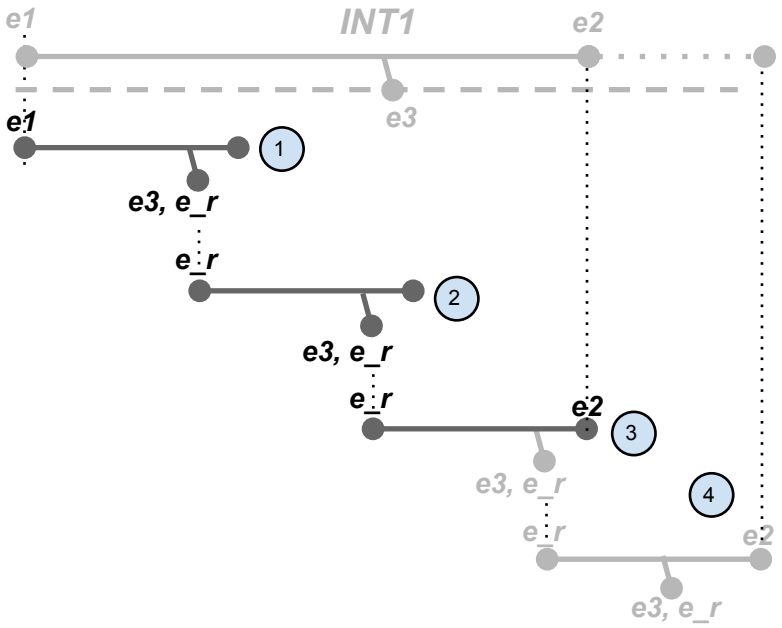

Figure 5.49: Interval *RTR* example scenario

In the *RTR* specification (5.31) we add the abort-trigger event e_r and extend the abstract index set X with a sub-interval index set Y.

$$\begin{aligned} &RTR(X \times Y, e1, e\_r;\ e2;\ e3, e\_r; \\ &\quad TP_1(DLY, C(RTR\_t_{DLY})), TP_2(DDL, C(RTR\_t_{DDL}))) \end{aligned} \tag{5.31}$$

Most of the Retry interval semantics is achieved by combining previously introduced templates; here we cover only the Retry transformation-specific aspects. Fully detailed generic templates and template calls are provided in section A.3. In principle, the Retry transformation reuses the same variable definitions and consistency invariants as the Single-to-Multi transformation. The key difference is the definition of when the Retry interval sequence is considered as responded to or aborted.

We display interval *RTR* schematic in Figure 5.50. All abstract interval variables are replaced with the new ones. Additional auxiliary variables *RTR_trig_intr* and *RTR_retry* are added.

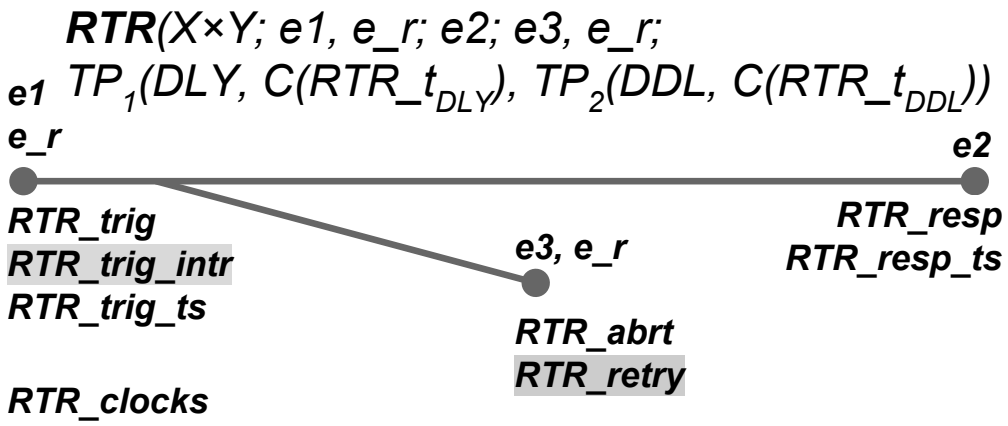

Figure 5.50: Interval *STM*

The *RTR*-used variables (Figure 5.50) are created from the reused *ROOR* base template. Auxiliary variable *RTR_trig_intr* type and purpose are identical to that of in the Single-to-Multi approach. The new auxiliary variable *RTR_retry* subsets interval's abort variable *INT*1_*abrt*. It records interval scope aborts that abort the whole sequence (event *e*3) and local aborts that abort just a sub-instance (abort-trigger event *e_r*). Further refinements of the interval will be equally affected by both abort types, hence the reason we record all abort type indices in the same variable.

There are four behaviour-defining invariants that are specific to the Retry refinement transformation:

i1 – every active *RTR* sequence corresponds to an active abstract interval instance.

i2 – a responded to interval *RTR* sub-instance corresponds to a responded abstract interval instance.

i3 glues the abstract and concrete abort indices. Note that the invariant filters out *RTR_retry* values. Auxiliary variable *RTR_retry* records abort-trigger event *e_r* occurrences that are local to a sub-interval and do not exist in the abstract interval. The fourth invariant ensures that there can be no running sub-instances for the sequence that has been responded to.

i4 – if the *RTR* sequence has been responded to, then none of its corresponding sub-instances may be active.

The three invariants are analogous to *STM* base template invariants *STM_consist_i*4-6 respectively (Figure 5.40).

INVARIANTS
Invs
RTR_type_1 : RTR_trig_intr ⊆ RTR_trig
RTR_type_2 : RTR_retry ⊆ RTR_abrt
RTR_consist_i1 : dom(RTR_trig) = INT1_trig
RTR_consist_i2 : dom(RTR_resp) = INT1_resp
RTR_consist_i3 : ∀x·x ∈ dom(RTR_abrt \ RTR_retry) ⇒ x ∈ INT1_abrt
RTR_consist_i4 : ∀x·x ∈ RTR_resp ⇒ dom({x}) ⊈ dom(RTR_trig \ (RTR_resp ∪ RTR_abrt))

Figure 5.51: RTR gluing invariants.

The refined *RTR* trigger and abort events behave in the analogous manner as the *ROOT* timing interval events (section 4.5): the refined trigger event $e1$ updates the *RTR* trigger variables with the new sub-instance index; the refined abort event $e3$ adds the sub-instance index to the *RTR* abort variable.

The behaviour of new abort-trigger event *e_r* (Figure 5.52) is achieved by reusing previously introduced templates. The abort role of the event is constructed from the abort template *TPL.ROOT.A* (Figure 4.25) and adds the semantics identical to that in abort event $e3$. The trigger role for the event is constructed similarly to interval *STM* trigger event $e2\_1$ (Figure 5.43). Additional guard *RTR_g*1 (Figure A.13) ensures that aborted and triggered *RTR* instances are synchronised, where *p_RTR_abrt* and *p_RTR_trig* are abort and response parameters generated by the abort and trigger templates respectively. Additional action *RTR_a*1 updates auxiliary variable *RTR_retry* with the aborted index. The variable is then used in invariant *RTR_consist_i*3 (Figure 5.51) to distinguish between local scope abort occurrence *e_r* – which aborts only *RTR* sub-instance and does not affect the abstract interval *INT*1 – and the interval scope abort $e3$ occurrence that affects the abstract interval.

Event *e_r* ≙
any p_RTR_abrt p_RTR_trig
where
Grds
RTR_g1 : dom(p_RTR_abrt) = dom({p_RTR_trig})
then
Acts
RTR_a1 : RTR_retry := RTR_retry ∪ p_RTR_abrt
end

Figure 5.52: Retry event template.

The refined response event $e2$ adds the sub-instance's index to the response variables. Additional guard *RTR_resp_g*1 (Figure 5.53) maintains invariant *RTR_consist_i*4 by ensuring that if the refined response event $e2$ responds to the *RTR* sequence then no sub-intervals for that sequence will remain active.

Event $e2$ refines $e2$ $\widehat{=}$

where

Grds

RTR_resp_g1 : $\mathtt{dom}(\{\mathtt{p\_RTR\_resp}\}) \lhd \mathtt{RTR\_trig} \setminus (\mathtt{RTR\_resp} \cup \mathtt{RTR\_abrt} \cup \{\mathtt{p\_RTR\_resp}\}) = \varnothing$

...

Figure 5.53: Response event template.

The deadline and delay invariants for the *RTR* interval are identical to those defined for the *STM* interval. The slight differences in the guards that ensure their preservation is reflected in the provided generic templates in subsection A.3.3 and subsection A.3.4. We reset *RTR* interval instance similarly to *STM* as shown in Figure 5.47.

5.1.5.1 Interval RTR dynamics

We display the RTR dynamics graph in Figure 5.54. The graph matches the scenario depicted in Figure 5.49. The RTR sequence is triggered by firing event $e1$. Abstract and concrete trigger variables are updated with the index. Note that abstract index X_1 matches concrete index domain $X_1 \mapsto Y_1$ ($s2$). At some point the instance is retried. The abort-trigger event e_r records the $X_1 \mapsto Y_1$ instance to abort and retry sets ($s3$). By recording it to the retry set we indicate that this abort event does not abort the sequence, but just one sub-instance. A new instance $X_1 \mapsto Y_2$ is triggered. We fire e_r once more, aborting the active instance and triggering a new one ($s4$). Response event $e2$ follows, responding to the active instance $X_1 \mapsto Y_3$ ($s5$). Event $e2$ also adds index X_1 to the abstract response variable.

For example, if the sequence is aborted by event $e3$ while in state $s3$, it adds the aborted sub-instance index to the abstract and concrete abort variables.

The reset event *INT*1*_rst* clears abstract and concrete indices that were responded to or aborted.

5.1.6 Parallel Refinements

Parallel refinement is a feature (rather than a transformation) that allows multiple refinement transformations to be applied on the same abstract timing interval at any refinement level. The refining intervals are synchronised via the refined events and shared

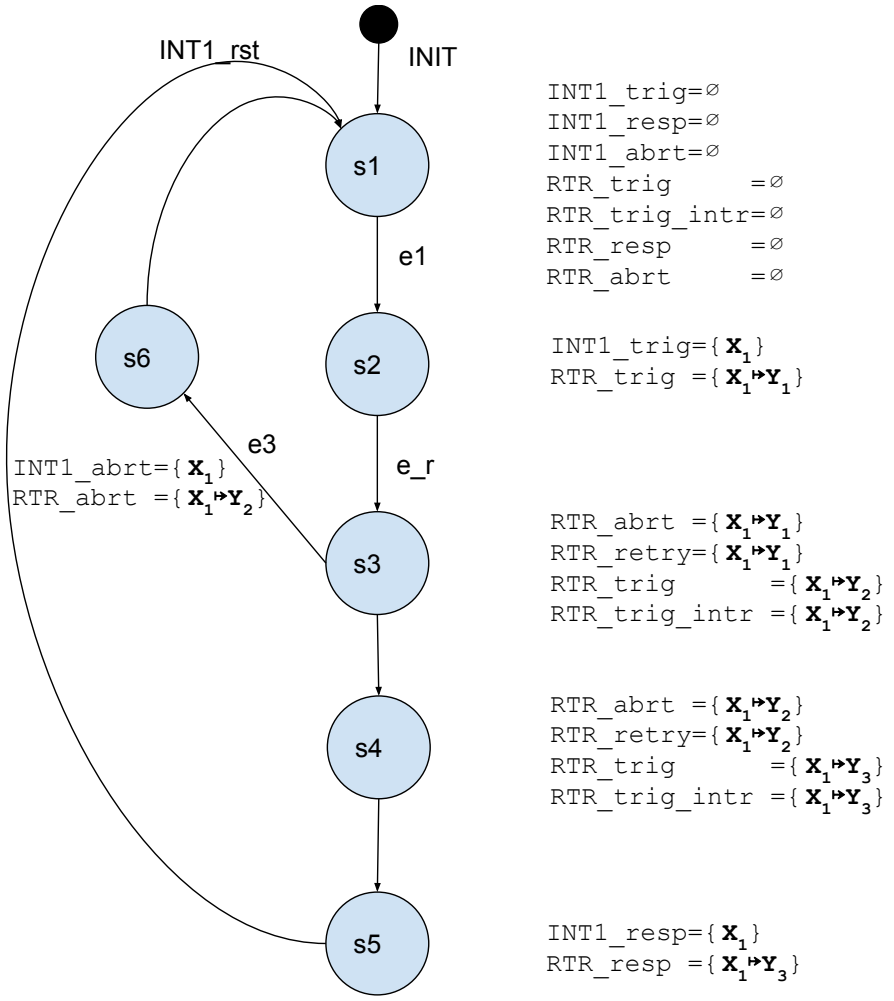

Figure 5.54: Retry interval *RTR* dynamics.

variables. The refining intervals can then be further elaborated. The feature itself does not involve specific Event-B constructs.

Consider an example machine $m0$ with interval $INT1$, defined in Figure 5.3. For simplicity, we omit abort event $e3$ and timing properties. The effect of parallel refinement on the abort event is analogous to other event types. The interval is triggered by event $e1$ and responded to by event $e2$. Figure 5.55 shows the interval in the refinement tree at the top left and its state machine representation on the right.

In the first refinement $m1$ we refine the interval into two parallel sequences of sub-intervals using the Sub-Interval refinement transformation (subsection 5.1.2). The first sequence comprises sub-intervals $SUB1$ and $SUB2$ with shared shared response-trigger event $e12$; the second comprises sub-intervals $SUB3$ and $SUB4$ with shared response trigger event $e34$.

The sequences reuse abstract trigger and response variables as displayed in Figure 5.56. Marked in bold are new variables, added by the Sub-Interval refinement transformation.

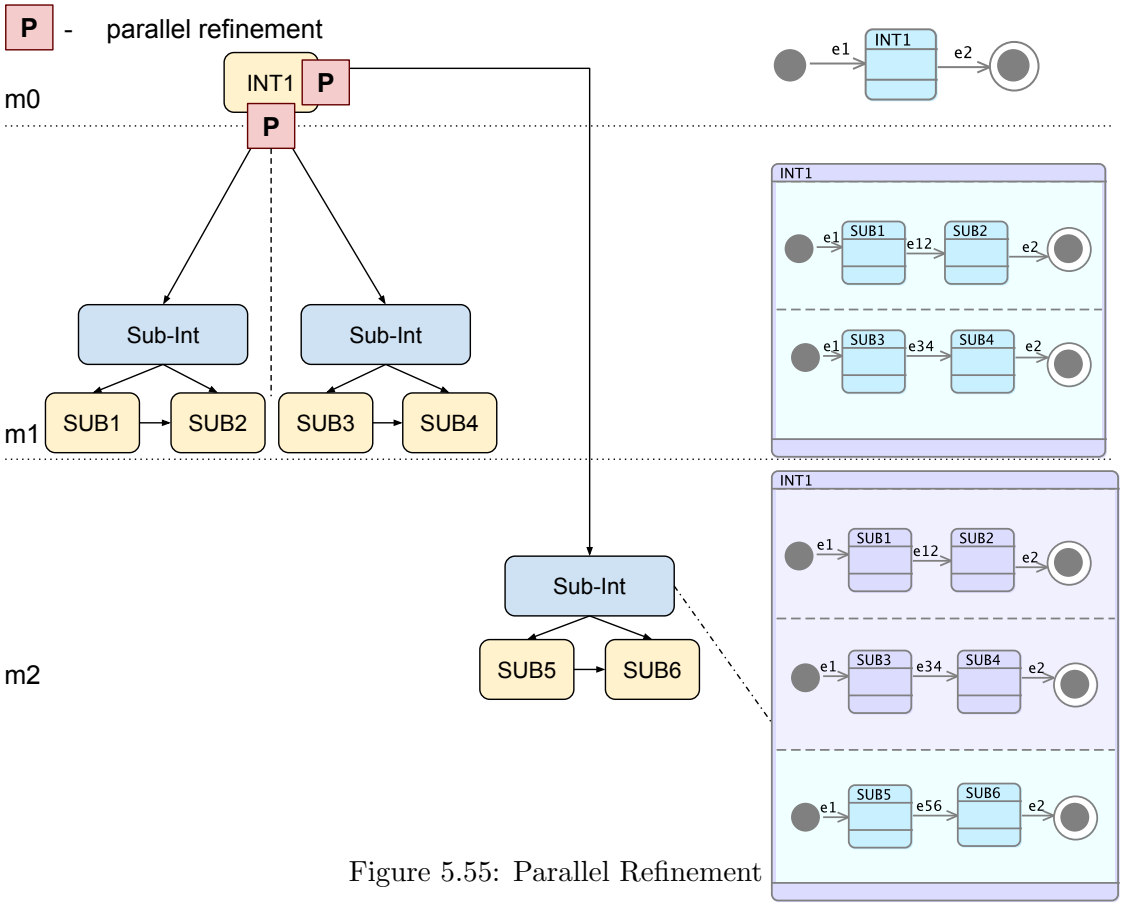

Figure 5.55: Parallel Refinement

The sequences are synchronised by the shared trigger $e1$ and response $e2$ events. Upon firing trigger event $e1$, both sequences are triggered with the same instance index which is stored in the shared trigger variable $INT1\_trig$. When the response event $e2$ occurs, both sequences are responded to and the interval instance index is recorded to the shared response variable $INT1\_resp$. Their index is X and their time flow is recorded in the same variable $INT1\_clocks$. Therefore their instance index and the time flow is always synchronised. The sub-intervals can have different timing properties; the number and occurrence times of the intermediate events is not constrained.

The abstract interval $INT1$ can be parallel-refined by any number or combination of the Sub-Interval, Alternative, Single-to-Multi and Retry refinement transformations.

We demonstrate how the abstract interval $INT1$ is refined again at a later stage. The refinement is feasible because we use superposition refinement and the abstract timing interval variables do not disappear. For example, in the second refinement $m2$ we refine $INT1$ to a third sequence of sub-intervals $SUB5$ and $SUB6$ with a shared response-trigger variable $e56$ (Figure 5.56). The sequence reuses the abstract clock $INT1\_clocks$ and the index set X. Sub-intervals $SUB5$ and $SUB6$ reuse corresponding abstract trigger and response variables.

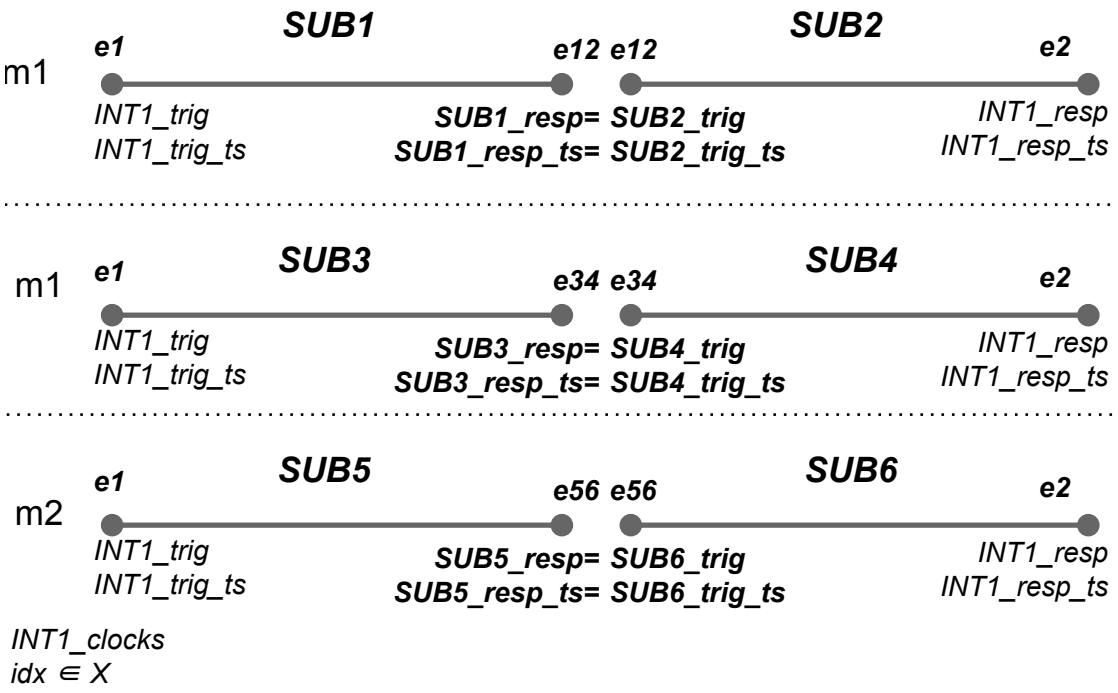

Figure 5.56: Sub-interval sequence variables.

The achieved behaviour is congruent with iUML diagramming rules when a state can be elaborated with an additional concurrent region at any refinement. The regions are then synchronised via common events. The use case of the parallel refinement feature is demonstrated in the pacemaker case study (chapter 9).

5.2 Data Refinement of the Timing Interval

Our attempt to data refine the timing interval was unsuccessful due to the inability to discharge clock related proof obligations. To explain the problem, we provide a schematic diagram that visualises data and superposition refinements of the timing interval $INT1$ side by side.

The principle of refining an interval to two sub-intervals using superposition refinement is explained in detail in subsection 5.1.2. We remind the reader that sub-interval $SUB1$ reuses abstract trigger variables and sub-interval $SUB2$ reuses abstract response variables. Both sub-intervals use the abstract clock variable $INT1\_clocks$. The newly introduced response variables for interval $SUB1$ and trigger variables for $SUB2$ have identical values. Newly introduced variables are marked in bold.

In the data refinement that we have attempted, $INT1$ related variables are replaced with a set of new trigger, response and clock variables for each of the sub-intervals: variables in bold with prefixes $SUB1\_$ and $SUB2\_$ correspond to sub-intervals $SUB1$ and $SUB2$ respectively.

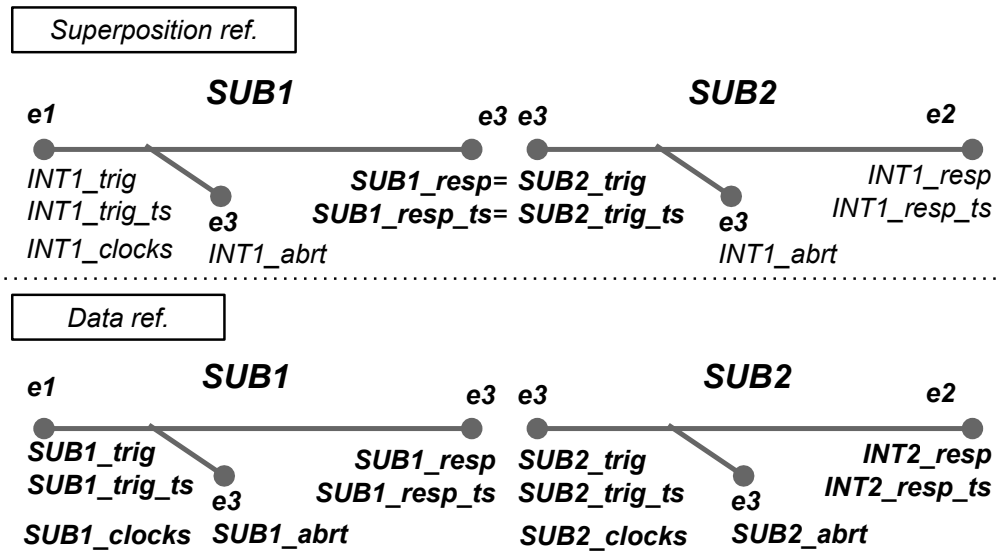

Figure 5.57: Sub-Interval superposition and data refinements.

The newly introduced variables are related to the refined abstract interval $INT1$ variables via gluing invariants. Abstract trigger variables match $SUB1$ trigger variables (5.32) and abstract response variables match $SUB2$ response variables (5.33). Response variables of the $SUB1$ interval must have the same values as the $SUB2$ trigger variables (5.34).

$$INT1\_trig = SUB1\_trig \qquad INT1\_trig\_ts = SUB1\_trig\_ts \tag{5.32}$$

$$INT1\_resp = SUB2\_resp \qquad INT1\_resp\_ts = SUB2\_resp\_ts \tag{5.33}$$

$$SUB1\_resp = SUB2\_trig \qquad SUB1\_resp\_ts = SUB2\_trig\_ts \tag{5.34}$$

The sub-interval $SUB1$ clock starts at zero and progresses until the interval instance is responded to. The $SUB2$ interval clock initialises with the value of the $SUB1$ clock. To ensure consistency between the abstract clock $INT1\_clocks$ and the concrete clocks $SUB1\_clocks$ and $SUB2\_clocks$ we use two invariants. The first invariant (5.35) matches the abstract clock with the $SUB1$ clock until the sub-interval is responded to. The second invariant (5.36) matches the abstract clock with the $SUB2$ clock.

$$\forall x \cdot x \in t1\_trig \land x \notin t1\_resp \Rightarrow t0\_clocks(x) = t1\_clocks(x) \tag{5.35}$$

$$\forall x \cdot x \in t1\_resp \land x \in t2\_trig \Rightarrow t0\_clocks(x) = t2\_clocks(x) \tag{5.36}$$

The abstract deadline guard in the *Tick* event is replaced with two concrete deadline guards (5.37) and (5.38) for $SUB1$ and $SUB2$ respectively.

$$\forall\, idx \cdot idx \in SUB1\_trig \wedge idx \notin SUB1\_resp \cup SUB1\_abrt \Rightarrow SUB1\_clocks(idx) + 1 \leq SUB1\_trig\_ts(idx) + SUB1\_DDL\_DUR \tag{5.37}$$

$$\forall\, idx \cdot idx \in SUB2\_trig \wedge idx \notin SUB2\_resp \cup SUB2\_abrt \Rightarrow SUB2\_clocks(idx) + 1 \leq SUB2\_trig\_ts(idx) + SUB2\_DDL\_DUR \tag{5.38}$$

We were unable to prove that the guards (5.37) and (5.38) are at least as restrictive as the abstract guard. From the interactive prover, we have learned that the abstract clock variable $INT1\_clocks$ cannot be substituted with the concrete counterparts, implying that the gluing invariants (5.35) and (5.36) are too weak. This motivated us to use a single abstract clock the timing interval and all its refinements.

One way to overcome this proof difficulty is to use a global time variable shared by all timing intervals. The variable then could not be upper-bounded, rendering full-state coverage with finite model-checkers impossible.

5.3 Overview and Conclusions

In this chapter we have presented five compositional refinement transformations: Sub-Interval, Alternative, Abort-to-Response, Single-to-Multi and Retry. The transformations were demonstrated on a simple example. For each of them, we have provided generative templates and described the transformation process and semantics. The generative approach allows for process automation and prover optimisations, both of which are discussed in chapter 6 and section 8.1 respectively.

We believe that the principle of template-based modelling can be applied to the functional part of the model, providing the same advantages as the timing interval approach: that is, potential auto-prover optimisations and modelling automation .

The refinement transformations have been validated and verified in three case studies in chapter 9, chapter 10 and chapter 11.

Chapter 6

Timing Interval Automation

In this chapter, we introduce the timing interval modelling automation tool.

6.1 tiGen Tool

As part of this research, we have developed a plug-in to automate the modelling of timing interval and its refinement transformations. This tool has been tested in the pacemaker, message passing protocol and landing gear system case studies in chapters 9, 10 and 11 respectively.

The plug-in builds on the iUML framework [11]. The framework is being used for class and state machine diagram modelling. The framework provides a diagram-to-Event-B syntax generation mechanism and an application programming interface (API) for reusable functionality, such as diagram element editors. Although modifications and extensions were needed, the framework significantly facilitated the development of the plug-in.

The plug-in provides the following features:

- **Support for timing interval and its refinement patterns.** The plug-in can add modify or elaborate an already existing timing interval, or it can add a new one.
- **Visual representation of timing intervals and their relations.** Timing intervals, elaborations and their relations are visually represented in a diagram. We have chosen a visual representation of the timing interval and its refinement transformations. The diagrammatic approach better represents the timing interval refinement structure.

- **Standalone functionality and compatibility with iUML.** The tool is designed to work as a standalone. Both tools can be used in tandem as discussed in chapter 7. The integration of tiGen with iUML is part of the future work described in section 12.8.
- **Validation prior code generation.** The plug-in partially validates the timing interval configuration specified by the user.
- **Refinement support.** The tool automates timing interval refinement transformations. The tool can handle event name changes and can automatically reconstruct a timing interval in the concrete level based on the abstract level. The tool facilitates the addition of the timing interval to an already existing model at any refinement level of the model.

6.1.1 What is Not Generated

Mostly due to time constraints, some aspects were left out and had to be performed manually:

- The tool generates only some of the dynamic duration consistency invariants between the abstract and concrete timing intervals.
- The timing interval currently supports only non-strict inequality between the timing property durations, e.g. $DLY_t \leq DDL_t$ – the durations in principle can be equal.
- Validation is not entirely implemented.
- Not all expiry-related consistency invariants are automatically generated.
- There is no integration with iUML. Regarding visual representation and code generation, these aspects are partially covered in chapter 7.
- Various bugs may be present in the plug-in and a more thorough testing is required.

6.1.2 User Interface

We briefly overview the interface of the tool. Timing interval related elements are displayed in a diagram called *Timing Interval Diagram* (TID). To add a new TID, the user has to right-click on an already existing machine and select option "Add Timing Interval Diagram" (Figure 6.1). When added, the TID will appear as a child of the corresponding machine. Figure 6.2 illustrates the added TID named *demoTID*. A machine can have many TIDs.

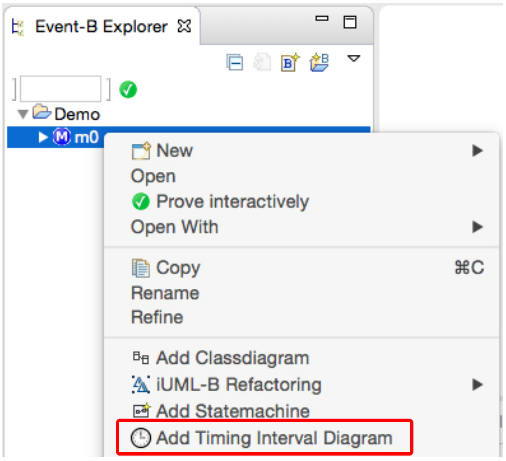

Figure 6.1: Add new TID.

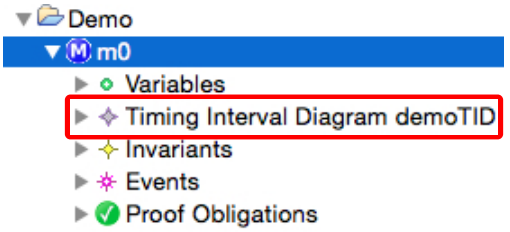

Figure 6.2: Added new TID.

TID's main working window (Figure 6.3) comprises the timing interval diagram canvas (1) and the toolbox palette. The palette provides the following elements: Timing Interval (2), a list of available refinement transformations (3) and a connector to relate the elements (4).

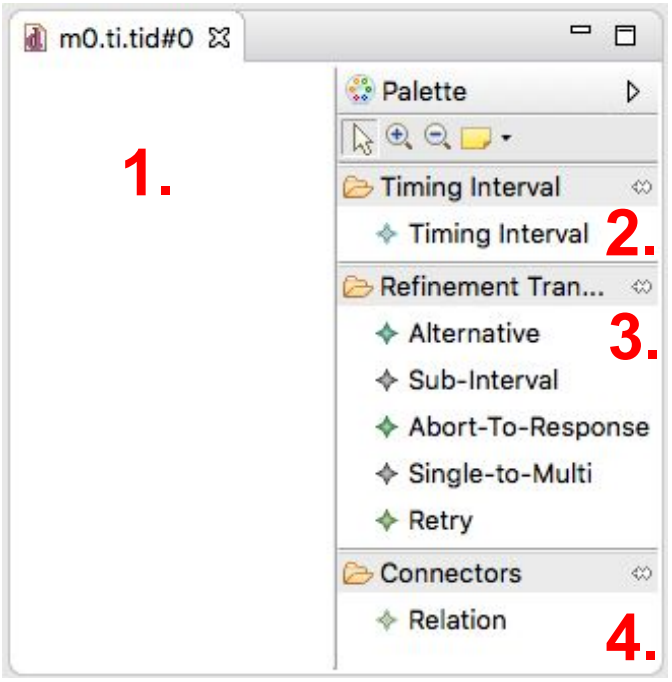

Figure 6.3: Timing interval canvas and element palette.

The TID can contain a number of timing intervals, which can be related by connectors. Clicking anywhere on the canvas opens a properties menu with the option to change the canvas name, link it to the abstract canvas (refine) and configure a *Tick* event for the timing intervals contained in this TID to use.

6.1.3 Adding New Timing Interval.

A new timing interval (TI) is added by selecting the TI element from the toolbox (2), displayed in Figure 6.3, and dragging and dropping it onto the diagram canvas (1). We provide an example view and specification of an already configured TI named *demo* Figure 6.4.

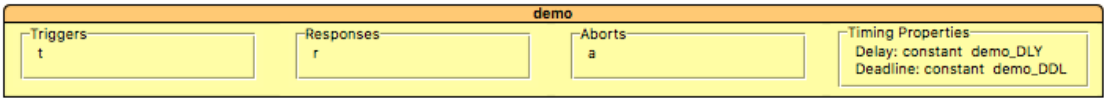

Figure 6.4: Timing interval *demo* representation in tiGen
$demo(X;\ t;\ r;\ a;\ demo\_DLY_{DLY}, demo\_DDL_{DDL})$

The interval uses index set X, is triggered by a trigger event t, responded to by event r and aborted by event a. Delay and deadline timing properties are specified as constants *demo_DLY* and *demo_DDL*. Note that in the specification we add timing property classifiers to specify which duration belongs to which timing property (Figure 6.4). The tool does not automatically create the timing property constants.

The added TI is configured via the properties window (Figure 6.5). We list notable properties that the modeller can configure: (1) specifies the TI's name, e.g. *demo* in the given example; (2) picks an abstract TI to refine; (3) specifies a the TI instance index set, e.g. X; (4) by default TI instance index is taken from a set. By ticking the checkbox, the modeller tells the tool that the index set is an expression, e.g. Cartesian product of two sets $X \times Y$; (5) specifies the target context file, to which the tool will generate static Event-B elements. Other menu tabs allow the modeller to manage trigger, response and abort events, edit the timing properties list and configure index reset event.

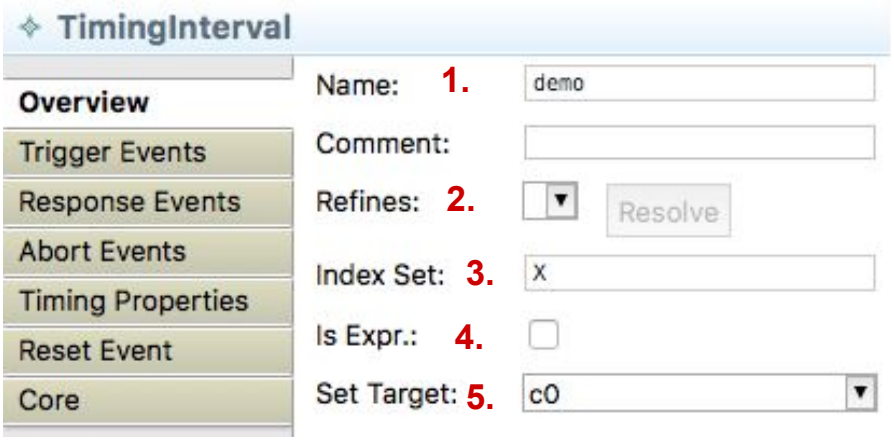

Figure 6.5: Timing interval properties window.

A resolve button near the refinement property (2) is a convenient feature. It automatically synchronises specification elements (events, timing properties) and the configuration properties between the abstract TI with the refining TI. In case of changes in the

model, the algorithm finds all concrete events that are related to the abstract TI and links all of them to the concrete TI. We give more detail on this in section 6.1.4.

The tab named *Core* (Figure 6.5) contains more advanced properties. Under it the modeller can access a property named *Type* that by default is set to *generic*, meaning that the tiGen tool will generate the TI as is. A *single-instance* option can be selected to constrain the generic approach to one interval instance at most. We discuss the Event-B semantics of this in section 4.7.

The trigger event t for interval *demo* is added via a dedicated properties window (Figure 6.6). The window provides convenient functions to manipulate the events: (1) link or unlink (2) an existing event to the specification; (3) create a new event and then link it; (4) unlink and delete event from Event-B machine; add a refinement (5) and remove (6) the refinement. The configuration window is standard for all event-related properties, e.g. response, abort or a *Tick* event properties. The added event names are reflected as lists of values in the corresponding boxes of the TI element.

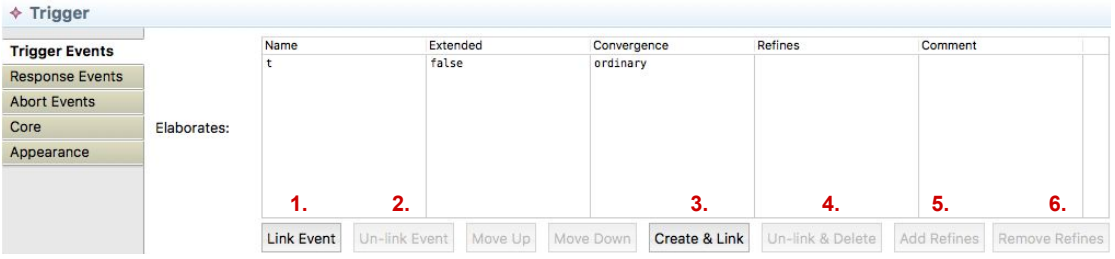

Figure 6.6: Timing interval trigger properties window.

A timing property is added via a similar widow as in Figure 6.6. Each timing property is then configured individually. The configuration window (Figure 6.7) configures the following properties: (1) timing property's kind (Deadline, Delay or Expiry); (2) duration type – either *dynamic* or *static*; (3) duration value. The latter can be set to an explicit natural number value or an expression, e.g. a constant like in our *demo* example. In case of the dynamic duration, the duration expression is left blank and is specified manually in the trigger event by the modeller.

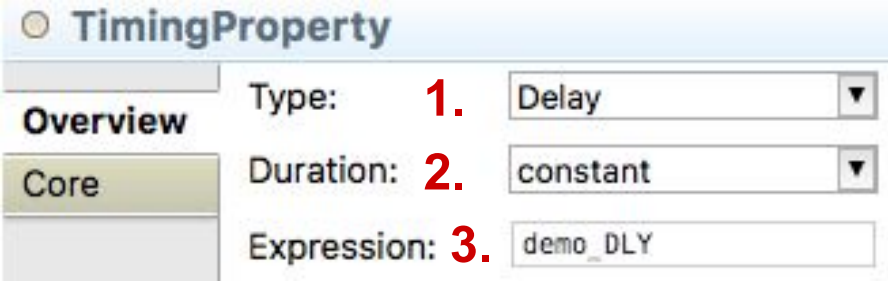

Figure 6.7: Timing interval trigger properties window.

6.1.4 Elaborating the Timing Interval

Existing TIs can be elaborated using one of the available transformations: alternative, sub-interval, abort-to-response, retry and parallel (chapter 5). Elaboration elements are listed in the diagram toolbox (3), in Figure 6.3. The elements are connected to TIs using a *Relation* element. The elaboration patterns can be applied on either the abstract or concrete TI. The TI element can be elaborated more than once. We explain the construction of the refinement transformations via a series of refinements $m1$ through to $m6$.

m1: Sub-Interval Refinement. Consider the example in Figure 6.8, where the *demo* TI is elaborated to three sub-intervals *Sub*_1, *Sub*_2 and *Sub*_3. The abstract timing interval is firstly connected to the refinement transformation to indicate what transformation is going to be performed. The transformation node is then connected to all sub-intervals, participating in the transformation. Sub-interval occurrence order is defined by connecting them together in a sequence.

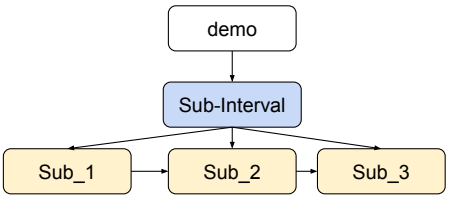

Figure 6.8: Timing interval *demo* refined to three sub-intervals.

The *Sub*_1 refinement is triggered (Figure 6.9) by the abstract trigger event t and responded to by the newly added response-trigger event $rt1$. All three sub-intervals can be aborted by event a. The timing property durations for this interval are set to zero. For every timing property with an explicit value, the tool automatically creates a TI-specific constant assigned that value, e.g. for the *Sub*_1 delay timing property it generates constant $Sub\_1\_DLY\_DUR = 0$.

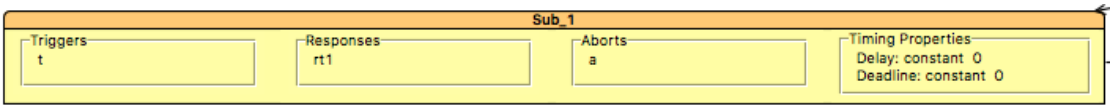

Figure 6.9: Sub-Interval *Sub*_1 representation in tiGen
$Sub\_1(X;\ t;\ rt1;\ a;\ TP_1(DLY, C(0)), TP_2(DDL, C(0)))$

We display sub-interval *Sub*_2 representation in the tiGen tool in Figure 6.10. The interval is triggered by response-trigger event $rt1$, which serves as a response event for sub-interval *Sub*_1. The example *sub*_2 is responded to by event $rt2$, which immediately triggers *Sub*_3. We have specified explicit timing property durations: the delay

and deadline are both equal to two time units. Unless otherwise stated, the delay durations will remain the same for all refinement transformations throughout the example refinements.

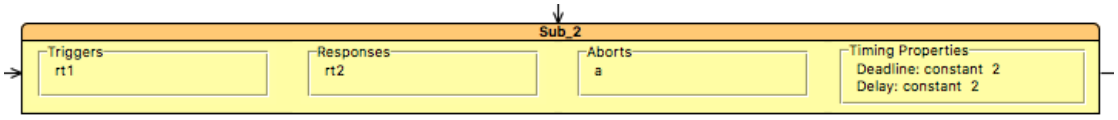

Figure 6.10: Sub-interval *Sub*_2 representation in tiGen
$Sub\_2(X;\ rt1;\ rt2;\ a;\ TP_1(DLY, C(2)), TP_2(DDL, C(2)))$

The *Sub*3 sub-interval (Figure 6.11) is responded to by the abstract event r. Both timing property durations are equal to zero. Note that the sub-interval sequence is triggered and responded to by the abstract interval *demo* events.

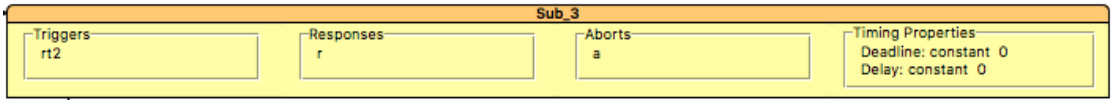

Figure 6.11: Sub-Interval *Sub*_3 representation in tiGen
$Sub\_3(X;\ rt2;\ r;\ a;\ TP_1(DLY, C(0)), TP_2(DDL, C(0)))$

m2: Alternative Refinement. In the next refinement we refine sub-interval *Sub*_2 into three alternative intervals *Alt*_1, *Alt*_2 and *Alt*_3 (Figure 6.12). Abstract interval *demo* and the connectors are not displayed in the figure.

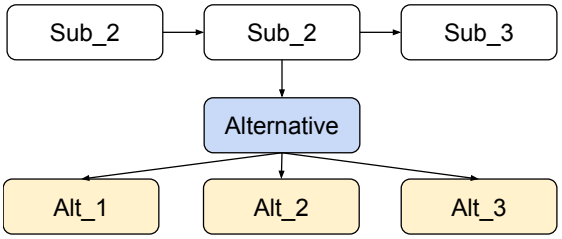

Figure 6.12: Sub-Interval *Sub*_2 refined into two alternative timing intervals.

Figure 6.13 is the tiGen representation of alternative interval *Alt*_1. Interval *Alt*_1 is triggered by event *rt*1_1 and responded to by *rt*2_1. Similarly, events *rt*1_2 and *rt*2_2 trigger and respond to interval *Alt*_2; events *rt*1_3 and *rt*2_3 trigger and respond to interval *Alt*_3. The events refine abstract counterparts *rt*1 and *rt*2 which are then removed from the refinement.

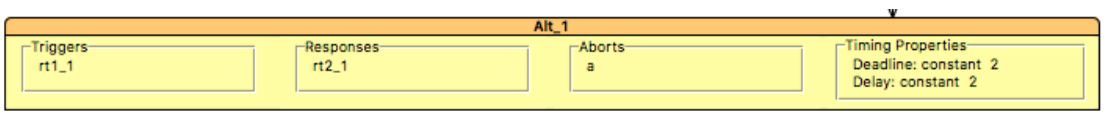

Figure 6.13: Alternative interval *Alt*_1
$Alt\_1(X;\ rt1\_1;\ rt2\_1;\ a;\ TP_1(DLY, C(2)), TP_2(DDL, C(2)))$

Due to the changes in the model, the tool updates the abstract sub-interval *Sub_2* specification (Figure 6.14). The tool updates the references to trigger and response events. As mentioned before, abstract events *rt*1 and *rt*2 were replaced with the concrete ones.

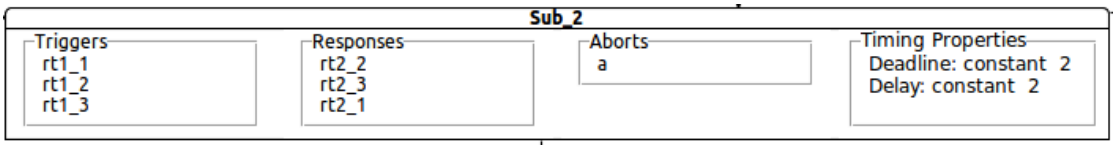

Figure 6.14: Updated abstract interval *Sub_2* specification: $Sub\_2(X;\ rt1\_1, rt1\_2, rt1\_3;\ rt2\_1, rt2\_2, rt2\_3;\ a;\ TP_1(DLY, C(2)), TP_2(DDL, C(2)))$

Abort-to-Response, Single-to-Multi and Retry Refinements. In the next three refinements – namely m3, m4 and m5 – we apply an Abort-to-Response refinement transformation on interval *Alt_1* (Figure 6.15), Single-to-Multi on *Alt_2* (Figure 6.16) and Retry on *Alt_3* (Figure 6.17). In terms of visual representation, all three refinement transformations are represented identically. The process of specifying parameters for all of the displayed transformations is similar to the others.

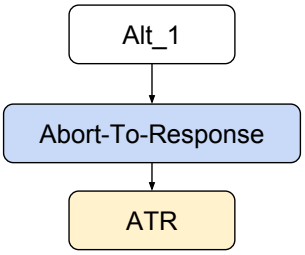

Figure 6.15: m3: interval *ATR* refining *Alt_1*.

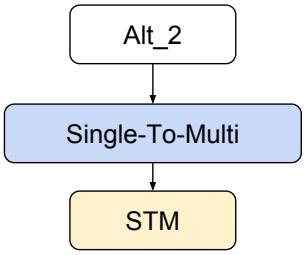

Figure 6.16: m4: interval *STM* refining *Alt_2*.

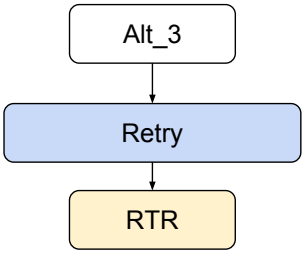

Figure 6.17: m5: interval *RTR* refining *Alt_3*.

m4: Parallelised Refinement. In the refined TID, the user can add a new TI or elaborate an already existing elements. Any TI, regardless of its position in the refinement tree, can be elaborated by any elaboration element.

In the final refinement, we show how to use the parallelised refinement feature to elaborate the most abstract timing interval *demo* with a sequence of the sub-intervals *Sub*_4 and *Sub*_5 (Figure 6.18). The parallel refinement feature is discussed in subsection 5.1.6.

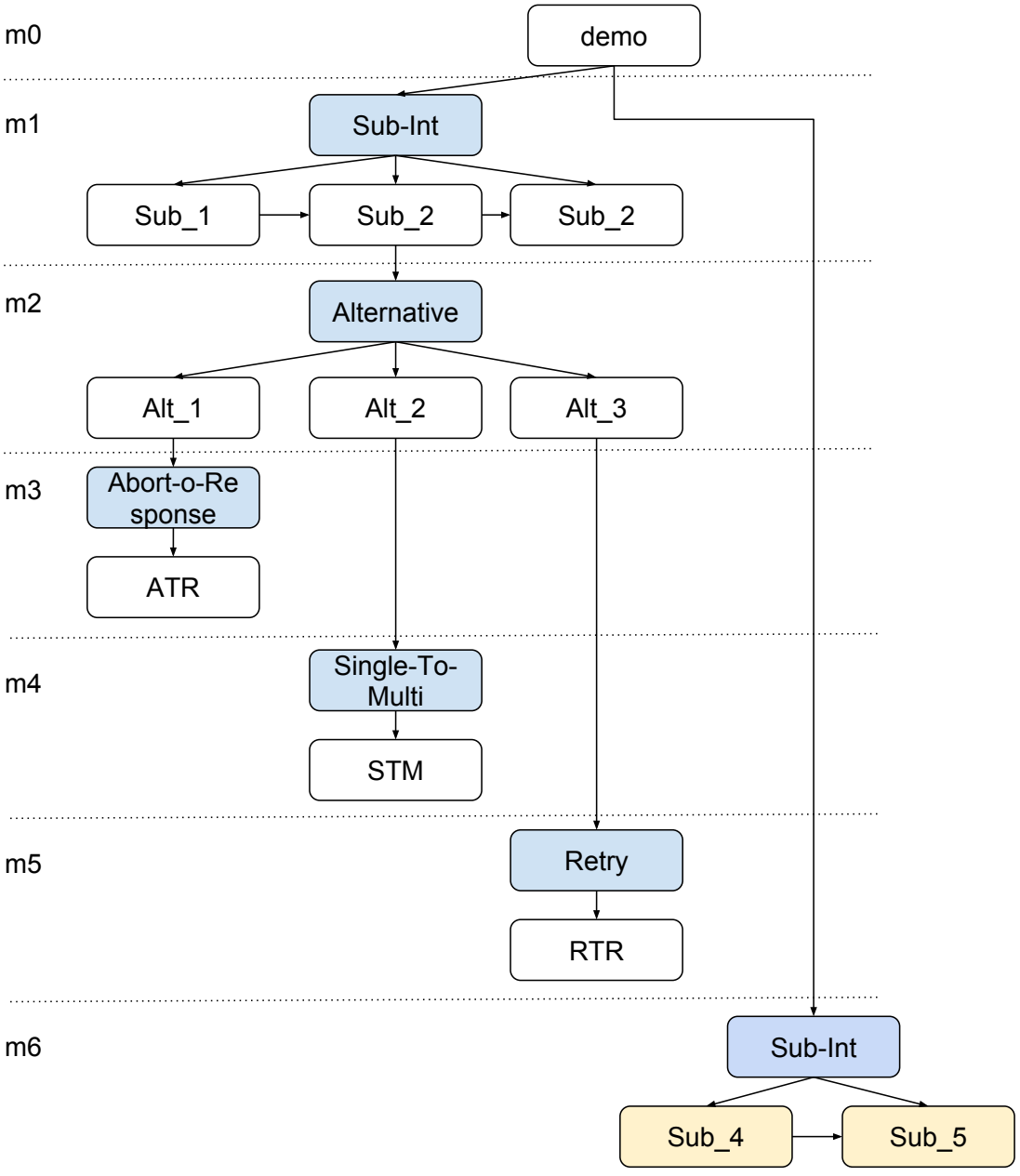

Figure 6.18: Possible elaboration point in the refinement.

Interval *Sub*_4 (Figure 6.19) is triggered by the abstract trigger event t and responded to by the response-trigger event $rt3$ that immediately triggers sub-interval *Sub*_5. All timing properties of both sub-intervals are equal to one time unit.

Active sub-interval *Sub*_5 is responded to by the abstract response event r (Figure 6.20).

The newly introduced sub-interval sequence and the abstract sequence of *Sub*_1, *Sub*_2 and *Sub*_3 sub-intervals are synchronised via shared events. Interval *Sub*_1 is triggered

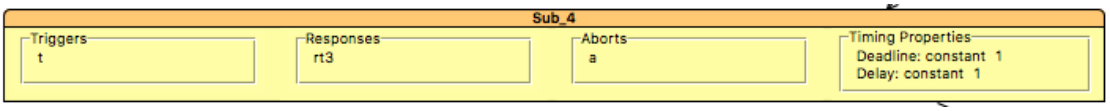

Figure 6.19: Sub-interval Sub_4:
$Sub\_4(X;\ t;\ rt3;\ a;\ TP_1(DLY, C(1)), TP_2(DDL, C(1)))$

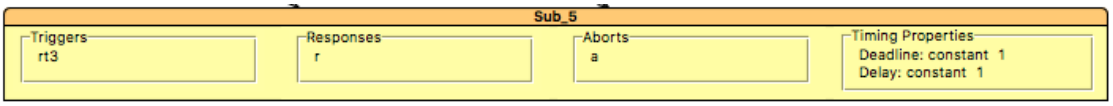

Figure 6.20: Sub-interval Sub_5:
$Sub\_5(X;\ rt3;\ r;\ a;\ TP_1(DLY, C(1)), TP_2(DDL, C(1)))$

by the same event t as interval Sub_4 (Figure 6.19); Sub_3 and Sub_5 (Figure 6.20) are responded to by the shared response event r.

Diagram Elements. The Event-B code is generated only for the elements that are visible in the diagram. Therefore, if the element is present in the abstract refinement and is absent in the concrete one, it will be excluded from the code generation. This is potentially useful if the timing interval is to be refined away. However, data refinement is beyond the scope of this thesis. We have left this for the future work discussed in section 12.8.

6.2 Overview and Conclusions

In this chapter, we have presented the tiGen tool for timing interval and its refinement transformation automation. The tool has been successfully applied in three case studies: the pacemaker (chapter 9), the message passing protocol (chapter 10) and the landing gear system (chapter 11).

Chapter 7

Modelling Guidelines

In this chapter we provide guidelines for modelling timing interval and its refinement transformation in tandem with state machine diagrams. The guidelines reflect theoretical and practical aspects: modelling tips, tool constraints and usability.

7.1 Modelling Work-Flow

In this thesis, we use iUML state machine diagrams to express the functional aspects of a system, and we use timing interval to model timing requirements. A smooth combination of the two requires compatible modelling techniques.

We recommend splitting modelling into two stages. Figure 7.1 depicts the workflow of how the timing interval is introduced into the model. The solid line marks the flow of events in the direction of the arrow. At first, the modeller should start by adding functional requirements. Fault handling is part of the functional aspect and therefore should be included. The timing interval relies on the already existing model elements. Therefore, timing requirements, if such are given, should be reflected in the functional model but without explicit time durations. For example, if there is a timing requirement such as "event B should happen within time t of the event A occurrence", the functional aspect should incorporate event A and B and ensure that event B eventually occurs. Time t is omitted. In further sections we provide recommended graphical representations of timing interval in state machine diagrams. The recommendations simplify the second stage of the modelling process.

The final refinement mN should have all the functionality except the explicit timing. A model with only functional aspects is simpler to verify and model-check. F found inconsistencies are less laborious to address Verification and validation, including manual animation with ProB, should be performed.

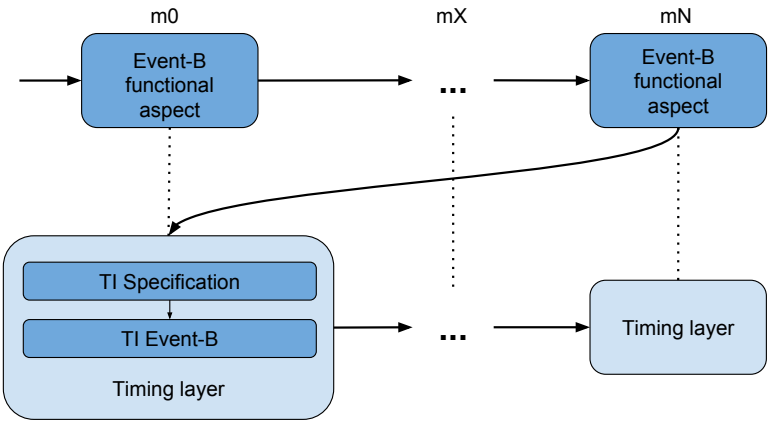

Figure 7.1: Modelling workflow.

Having done with the functional model, the modeller should reiterate through the functional model refinements and add timing gradually at the most abstract level possible. In case a machine becomes too complex, a number of separate timed-machine refinements can be inserted. Finally, the model should be verified and validated again with the added explicit time.

7.2 Mapping Timing Interval to State-Machine

In this section, we provide a recommended convention to represent a timing interval in an iUML state machine. We have the discussion on a given timing interval example *demo* (7.1). As described in the workflow section 7.1, this and the further given example state machine diagrams should come in the first modelling stage. Then, the actual timing interval generated with tiGen should follow. Timing interval *demo* and the subsequent refinements thereof are identical to those described in chapter 6.

The timing interval has one trigger event t and one response event r. Abort event a is present in this and future examples unless stated otherwise. Delay and deadline timing properties are set to constants $demo\_DLY_{DLY}$ and $demo\_DDL_{DDL}$ respectively. The example timing interval has a set of instances – X.

$$demo(X;\ t;\ r;\ a;\ TP_1(DLY, C(demo\_DLY_{DLY})), TP_2(DDL, C(demo\_DDL_{DDL}))) \tag{7.1}$$

Mapping can be split into two parts. The first part is to map the timing interval to the state machine graphically. The second part is to ensure that timing interval and state machine instances are synchronised in terms of active instances. The mapping occurs between iUML variables and the timing interval trigger, response and abort variables. Therefore, mapping is not affected by the timing property configuration.

Graphical Mapping. We demonstrate how the *demo* timing interval (7.2) and its events can be graphically represented by state machine *sm* (Figure 7.2). The given example is a recommended convention, and we acknowledge that it is not always possible to express timing interval in state machine diagrams in an unambiguous way.

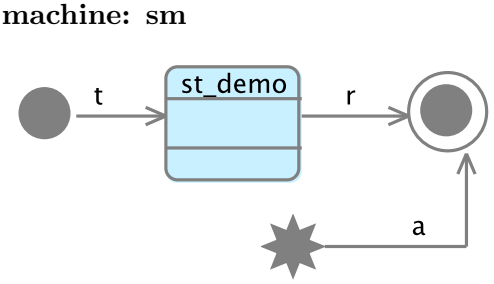

Figure 7.2: Abstract state machine for timing interval.

$$demo(X;\ t;\ r;\ a;\ TP_1(DLY, C(demo\_DLY_{DLY})), TP_2(DDL, C(demo\_DDL_{DDL}))) \tag{7.2}$$

We represent a timing interval with a state . Our example state machine *SM* has one state *st_demo* that represents the example timing interval *demo*. Typically the state is called exactly like the timing interval or has a prefix *st_*.

Trigger events are represented as transitions, whose target state represents the timing interval in question. In this example, trigger event t is represented as a transition with the target state *st_demo*. The source state of such transition can be any element. In this example, it is an *Initial* pseudo-state.

Similarly, the response event is represented by a transition whose source state is the timing interval in question. Such transition's target state is unrestricted. In this example, the response is represented by transition r, coming out from state *st_demo*.

We recommend representing abort events as a transition from pseudo-state *Any*. The pseudo-state does not constrain the transition.

Event-B Mapping. The mapping between the timing interval and the state machine at the Event-B level can work with the variable and enumerated translations of the state machine. In this particular example, we translate state machine *sm* to an enumerated set. We explain the semantics of the translation in detail in subsection 3.5.1. The state machine is semantically expressed as function *sm* (7.3). Set *sm_STATES* holds possible state machine instance states: either state *st_ti* or *NULL*. The former is a visually expressed state that is always active when the state machine is active. The *NULL* state denotes the fact that the state machine has not yet been initialised, hence it not running.

$$sm \in X \rightarrow sm\_STATES \tag{7.3}$$

We briefly explain the dynamics of the given state machine. Upon execution, event t sets a specific state machine instance to state st_demo:

$$sm(self) := st\_demo \tag{7.4}$$

Where *self* is the event parameter with the state machine instance in question. Similarly, events r and a set the specific instance to state $NULL$.

We can then synchronise the state machine with the timing interval with invariant (7.5). Where $demo\_trig$, $demo\_resp$ and $demo\_abrt$ are the trigger, response and abort index variables for interval $demo$. The invariant specifies that for every active but not yet responded to or aborted interval $demo$ instance there is a state machine instance with an active state st_demo. The equivalence requires the reverse to be true as well: if there is a state machine instance with an active state st_demo, then the timing interval $demo$ instance with the matching index must be active too.

$$\forall\, idx \cdot idx \in demo\_trig \setminus (demo\_resp \cup demo\_abrt) \Leftrightarrow sm(idx) = st\_demo \tag{7.5}$$

Special Case: Self-Loop. An interval can have an overloaded event that serves as the trigger and the response event for the timing interval. Consider a self-loop timing interval $demo2$ (7.6) that has a response-trigger event rt (Figure 7.3). An abort event is not specified. We visualise such an interval with state machine $sm2$ in Figure 7.3. Transition tr has the matching source and target state st_demo2. The state machine has a number of instances identified by the elements from set X.

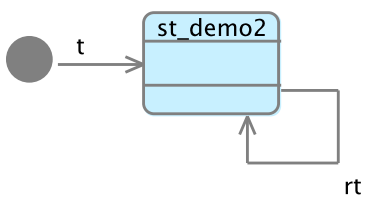

Figure 7.3: A self-loop timing interval $demo$, represented by state machine $sm2$.

$$\begin{aligned} &demo2(X \mapsto \{0,1\};\ rt;\ rt;\ --; \\ &\quad TP_1(DLY, C(demo2\_DLY_{DLY})), TP_2(DDL, C(demo2\_DDL_{DDL}))) \end{aligned} \tag{7.6}$$

We lift the interval $demo$ index set to a Cartesian product $X \mapsto \{0,1\}$. Before explaining the reasons for this, we demonstrate the dynamics of the state machine $sm2$ and interval $demo$ in a small animation Figure 7.4. For simplicity, we limit the cardinality of set X

to one. Variables *demo*2_*trig*, *demo*2_*resp* and *demo*2_*abrt* hold interval *demo* trigger, response and abort indices. Variable *st_demo*2 records indices of state machine instances that are in active state *st_demo*.

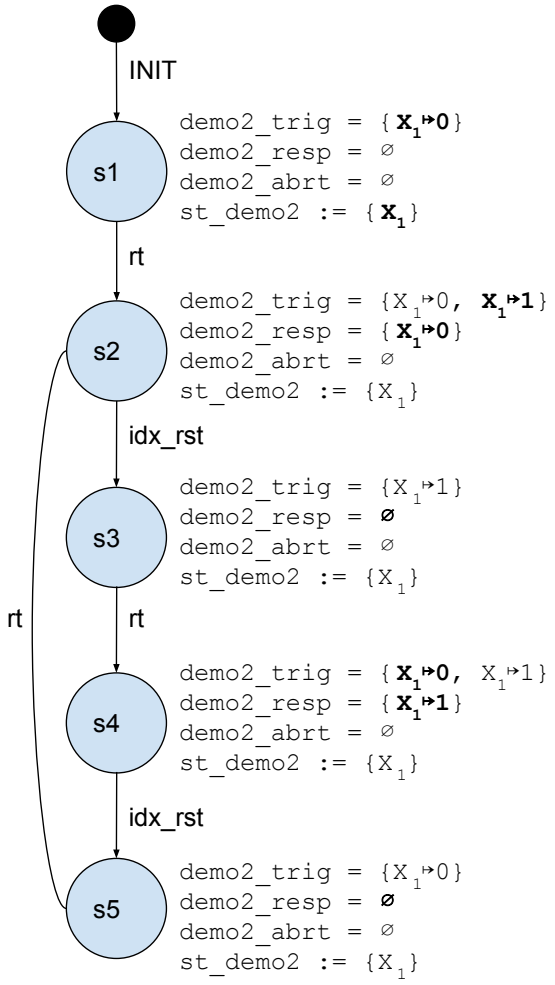

Figure 7.4: The dynamics of interval *demo* and state machine *sm*2

The model initialises in active state *st_demo*2 and the active interval with instance indices X_1 and $X_1 \mapsto 0$ respectively ($s1$). We highlight new and updated values in bold. When event *rt* fires, the active interval instance is responded to and the new instance $X_1 \mapsto 1$ is triggered ($s2$). Note that the state machine instance remains set to X_1 throughout the animation. A responded to index $X_1 \mapsto 0$ is cleared with the index reset event *idx_rst* ($s3$). Then, event *rt* fires again, responding to instance $X_1 \mapsto 1$ and triggering $X_1 \mapsto 0$ ($s4$). The responded index is cleared ($s5$) and the process repeats again ($s2$).

The two-dimension index set allows timing interval *demo* to: 1. stay synchronised with the state machine instance by index X; 2. alternate between unique interval instances 0 and 1. We express the synchronisation requirement between synchronised interval *demo*2 and state *st_demo*2 by set X with invariant (7.7). The invariant is analogous

to that of (7.5) except for the *dom*() operator that takes the domain value (X) of the active *demo* instance index.

$$\forall\, idx \cdot idx \in dom(demo\_trig \setminus (demo\_resp \cup demo\_abrt)) \Leftrightarrow sm(idx) = st\_demo \quad (7.7)$$

Second, we ensure that only one of the two possible ($\{0, 1\}$) instances can be active at any point in time. (7.8).

$$\begin{aligned} \forall\, idx \cdot idx \in dom(ti\_trig \setminus (ti\_resp \cup ti\_abrt)) \Rightarrow (\exists\, y \cdot y \in \{0,1\} \wedge \\ \{idx \mapsto y\} = \{idx\} \lhd ti\_trig \setminus (ti\_resp \cup ti\_abrt)) \end{aligned} \quad (7.8)$$

The special self-loop case applies only to the most abstract interval and never to its refinements. Therefore the two invariants (7.7) and (7.8) are applied only when two conditions are met: 1. it is a *root* (non-refining) timing interval; 2. the interval has a self-loop transition.

7.3 Mapping TI Refinement to a State Machine

We suggest a convention for visually representing timing interval refinement patterns in state machine diagrams. The convention is explained in interval *demo* (Figure 7.2) through a series of refinement steps. The refinement structure and timing intervals are identical to those in subsection 6.1.4.

m1: Sub-Interval. In the first refinement we demonstrate how a refinement of three sub-intervals *Sub*_1 (7.9), *Sub*_2 (7.10) and *Sub*_3 (7.11) can be represented in a state machine diagram (Figure 7.5). The specifications of the sub-intervals are identical to examples given in the Timing Interval Automation chapter figures: Figure 6.9, Figure 6.10 and Figure 6.11. The sub-intervals are represented in a region named *st_demo_SM*. The region contains three states, each named after a sub-interval with a prefix *st_*. E.g. state *st_Sub*_1 represents sub-interval *Sub*_1. Similarly, we map other two sub-intervals and states.

The first interval *Sub*_1 is triggered by event t that creates the state machine. The subsequent sub-intervals *Sub*_2 and *Sub*_3 are triggered by $rt1$ and $rt2$ events respectively. The last sub-interval is responded to by the same response event r that terminates the state machine instance. Abort event a can at any point in time abort any of the events. The states and the sub-intervals are then synchronised with invariants such as (7.5).

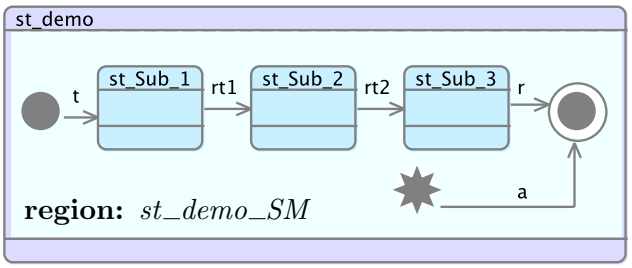

Figure 7.5: Sub-Interval representation in a state machine.

$$Sub\_1(X;\ t;\ rt1;\ a;\ TP_1(DLY, C(0)), TP_2(DDL, C(0))) \tag{7.9}$$
$$Sub\_2(X;\ rt1;\ rt2;\ a;\ TP_1(DLY, C(2)), TP_2(DDL, C(2))) \tag{7.10}$$
$$Sub\_3(X;\ rt2;\ r;\ a;\ TP_1(DLY, C(0)), TP_2(DDL, C(0))) \tag{7.11}$$

m2: Alternative. In the second refinement, we refine sub-interval *Sub*_2 to three alternative intervals *Alt*_1 (7.12), *Alt*_2 (7.13) and *Alt*_3 (7.14).

The intervals can be mapped with the state machine refinement as displayed in Figure 7.6. State *st_Sub*_2 is elaborated with a region, named *st_Sub*_2_*SM*. The region has three states mapping by names to the three alternative intervals. Each state is formally mapped to the corresponding alternative interval with the invariant (7.5).

Trigger event *rt*1 is refined to three alternative trigger events *rt*1_1, *rt*1_2 and *rt*1_3. The same applies to response events *rt*2_1 and *rt*2_2 that refine *rt*2_. The abort event *a* is visualised as a transition going from the *Any* state. The new state *Alt*_1 is synchronised with the timing interval *Alt*_1 via the invariant. The same is true with *Alt*_2 and *Alt*_3.

m3: Abort-to-Response. We apply the Abort-to-Response refinement transformation on alternative interval *Alt*_1 (7.15). This transforms abort event *a* for the interval to a response event. We remove *Any* state from the diagram and draw abort event *a* as a transition from the *Alt*_1 state to the *Final* pseudo-state (7.7). The event retains its role as the abort for alternative intervals *Alt*_2 and *Alt*_3.

This leads to the situation in which event *a* serves as the abort and as the response event for different intervals in the same state machine. The visual state machine notation is not expressive enough and creates ambiguity. In this situation it is not possible to distinguish between response and abort events. We recommend visually representing event *a* as an event outgoing from states *Alt*_1, *Alt*_2 and *Alt*_3 as in Figure 7.7.

m4: Single-to-Multi. In the fourth refinement, we demonstrate how a Single-to-Multi refinement transformation can be represented in a state machine diagram.

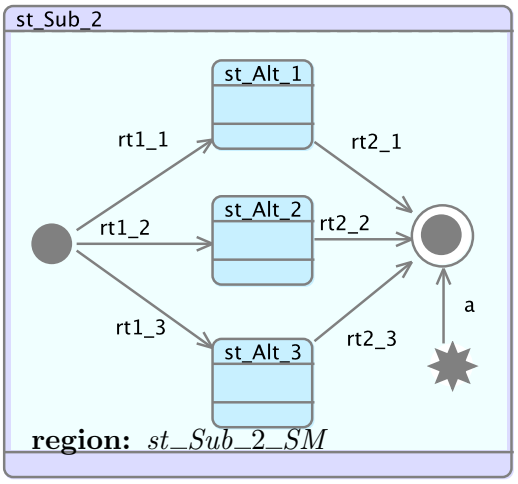

Figure 7.6: Alternative interval representation in a state machine.

$$Alt\_1(X;\ rt1\_1;\ rt2\_1;\ a;\ TP_1(DLY, C(2)), TP_2(DDL, C(2))) \tag{7.12}$$
$$Alt\_2(X;\ rt1\_2;\ rt2\_2;\ a;\ TP_1(DLY, C(2)), TP_2(DDL, C(2))) \tag{7.13}$$
$$Alt\_3(X;\ rt1\_3;\ rt2\_3;\ a;\ TP_1(DLY, C(2)), TP_2(DDL, C(2))) \tag{7.14}$$

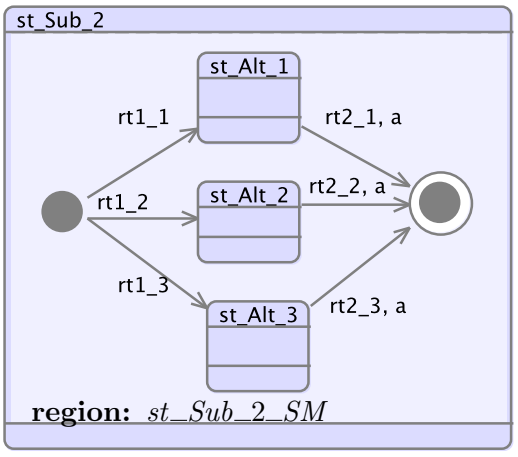

Figure 7.7: ATR (Alt_1), Alt_2 and Alt_3 representation in the state machine.

$$ATR(X;\ rt1\_1;\ rt2\_1, a;\ --;\ TP_1(DLY, C(0)), TP_2(DDL, C(0))) \tag{7.15}$$

For example purposes we refine alternative interval Alt_2 to interval STM by applying the Single-to-Multi refinement transformation (7.16). As described in subsection 5.1.4, the transformation requires at least one new trigger and one new response event – *stm_trig* and *stm_resp* respectively in this case. The interval index set is lifted to $X \times Y$. The timing property durations are set to constants STM_{DLY} and STM_{DDL}

We represent STM sub-instances in a new state machine *sm_STM* with one state

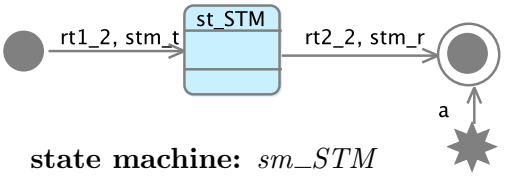

Figure 7.8: STM representation in a state machine.

$$STM(X \times Y;\ rt1\_2, stm\_t;\ rt2\_2, stm\_r;\ a;\\ TP_1(DLY, C(STM_{DLY})), TP_2(DDL, C(STM_{DDL}))) \tag{7.16}$$

st_STM (Figure 7.8). The state machine and the timing interval share the same index set $X \times Y$. Having matching index sets, we can map (7.5) each *STM* sub-instance to the state machine instance. Trigger events *rt*1_2 and *stm_t* are represented as a transition from pseudo-state *Init* to state *st_STM*, meaning that upon trigger event occurrence a new state machine instance, representing the *STM* sub-instance, is created. Similarly, response events lead to the *Final* pseudo-state, denoting that the state machine instance and the corresponding *STM* sub-instance are inactivated. We discuss the abort event later in the section.

We add a graphical representation of the *STM* sub-instance group. We elaborate state *Alt_2* with a region *sm_Alt*2_*STM* that has one state *st_Alt_2_STM* (Figure 7.9). The state represents an *STM* sub-instance group with at least one active sub-instance. The new events *stm_trig* and *stm_resp* are represented as self-loop transitions. Adding them to this region ensures the synchronisation of the events. The vents will be enabled only when state *st_Alt_1_STM* is active. These events trigger and respond to an active sub-instance of the *STM* interval, but they do not trigger or respond to the whole *STM* instance group. The refined response event *rt*2_2 denotes the response of the *STM* sub-instance group by terminating the state machine.

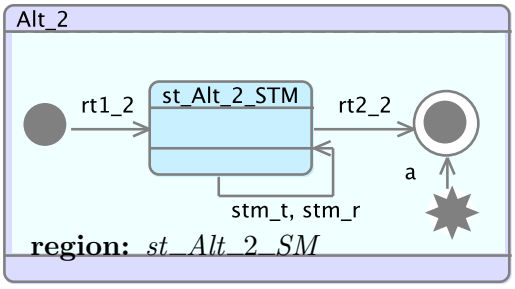

Figure 7.9: Abstract state machine for STM.

We ensure synchronisation between the *st_Alt_2_SM* region and state machine *sm_STM* with invariant (7.17). The invariant states that for every state machine *st_Alt_2_SM*

instance that is in active state *st_Alt_2_STM* there exists an active instance of *sm_STM* (Figure 7.8).

$$\forall\, st\_Alt2\_SM(d) = st\_Alt\_2\_STM \Leftrightarrow \exists\, d, r \cdot sm\_STM(d \mapsto r) = st\_STM \tag{7.17}$$

To summarise in an example, an instance X_1 of region *st_Alt2_SM* represents an active group of *STM* sub-instances (7.17) of type $\{X_1\} \times Y$. Each active sub-instance corresponds to an active instance of state machine *sm_STM* (7.5), e.g. *STM* sub-instance $X_1 \mapsto Y_1$ maps to the state machine instance with the same index.

Consider the case, when there are two group-related sub-instances running with indices $X_1 \mapsto Y_1$ and $X_1 \mapsto Y_2$. They are represented by the corresponding state machine *sm_STM* instances in active state *st_STM*. Region *st_Alt2_SM* for instance X_1 is then in active state *st_Alt_2_STM*:

$$st\_Alt2\_SM(X_1) = st\_Alt\_2\_STM \tag{7.18}$$

According to the defined behaviour in Single-to-Multi transformation (subsection 5.1.4), in case of the occurrence of the abort event $e3$, all group instances are aborted at once. In terms of state machine diagrams, this means that event $e3$ sets region instance $st\_Alt2\_SM(X_1)$ to inactive – *NULL*. Similarly, event $e3$ should inactivate instances $X_1 \mapsto Y_1$ and $X_1 \mapsto Y_2$ in state machine *sm_STM*. This is not the behaviour of iUML state machines. In iUML, the transition always terminates only one state machine instance. Therefore, by default only one of the two *sm_STM* instances would be inactivated. As a consequence, the consistency invariant (7.17) would then become invalidated. To address this, we modify the abort event $e3$ to abort both sub-instances.

m5: Retry. In the fifth refinement, we refine interval *Alt_3* to interval *RTR* (7.19) using the Retry refinement transformation. We add an abort-trigger event *rtr_at*, as required per instruction in subsection 5.1.5.

We represent each sub-instance of the *RTR* sequence in the new state machine *sm_RTR* (Figure 7.10). The *RTR* timing interval and state machine *sm_RTR* use the same instance index set $X \times Y$. Their instances are synchronised with invariant (7.5).

The abort-trigger event *rtr_at* in the diagram is represented as a transition from pseudo-state *Init* to state *st_Alt_2_STM* and as a transition from the state to pseudo-state *Final*. Upon execution the transition performs two tasks. Firstly, it creates a new *STM* sub-instance and its corresponding state machine instance. Secondly, it aborts the already existing *STM* sub-instance and inactivates the corresponding state machine instance.

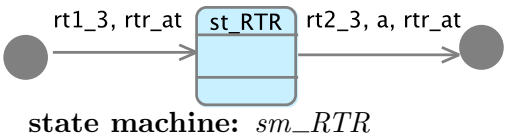

Figure 7.10: *RTR* sub-instance representation in state machine *sm_RTR*.

$$RTR(X \times Y;\ rt1\_2, rtr\_at;\ rt2\_2;\ a, rtr\_at; \\ TP_1(DLY, C(RTR_{DLY})), TP_2(DDL, C(RTR_{DDL}))) \tag{7.19}$$

We elaborate state *st_Alt_3* with region *st_Alt_3_SM*, containing one state *st_Alt_3_RTR* (Figure 7.11). The state represents an active *RTR* sequence.

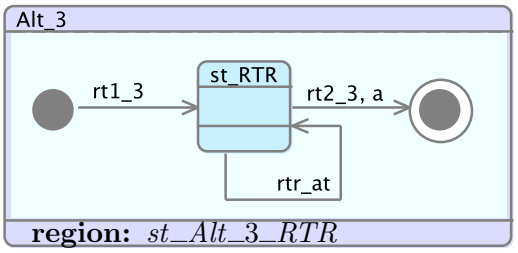

Figure 7.11: *RTR* sequence representation by state *st_Alt_3_RTR*.

Similar to the *STM* interval, the *RTR* timing interval is then synchronised with the *sm_RTR* state machine with invariant (7.20). The invariant requires that while the retry sequence is active, the state of the corresponding state machine must be active as well.

$$\forall\, st\_Alt3\_SM(d) = st\_Alt\_3\_RTR \Leftrightarrow \exists\, r \cdot sm\_RTR(d \mapsto r) = st\_RTR \tag{7.20}$$

m6: Parallel Refinement Feature. In the last refinement, $m6$, we refine the most abstract timing interval *demo* to two parallel sub-interval sequences *Sub_4* and *Sub_5* as shown in Figure 7.12. The full refinement tree is displayed in Figure 6.18. The refinement feature is explained in subsection 5.1.6.

The *st_demo* state already has a child state machine (Figure 7.13) representing the three sub-intervals introduced in refinement $m1$. We represent the sequence of sub-states *Sub_4* and *Sub_5* as a concurrent region under the parent state *st_demo*. The sub-intervals are then synchronised with the state machine with invariant (7.5).

The two sub-interval sequences are then synchronised by their trigger, abort and response events. This is as well reflected in the concurrent regions: both regions are initialised by trigger event t and destroyed by response event r and abort event a.

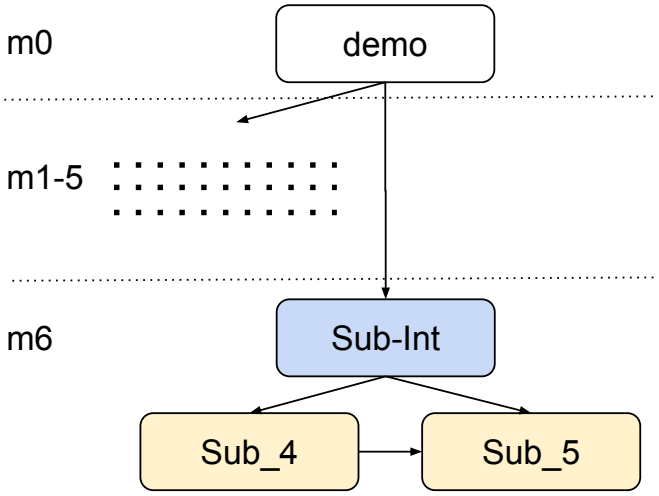

Figure 7.12: Parallel refinement

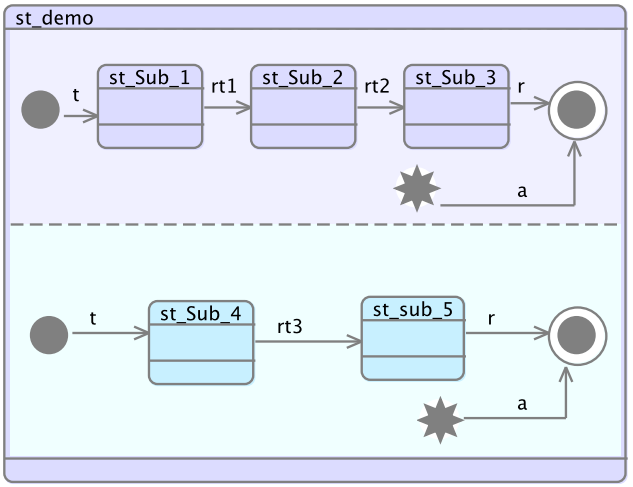

Figure 7.13: Parallel refinement representation in a state machine
$Sub\_4(X;\ t;\ rt3;\ a;\ TP_1(DLY, C(1)), TP_2(DDL, C(1)))$
$Sub\_5(X;\ rt3;\ r;\ a;\ TP_1(DLY, C(1)), TP_2(DDL, C(1)))$

7.4 Deadlock Verification with ProB

A deadlock may occur when there is more than one timing interval instance running in parallel and all of them share at least one response event. The instances may belong to two different timing intervals or be part of a Single-to-Multi interval. We assume that the functional part of the model is ideal and does not deadlock.

We demonstrate how a deadlock can occur between two parallel timing intervals based on the example model Figure 7.14, originally introduced in section 4.2. The example has two timing intervals $INT1$ and $INT2$ with deadline and delay timing properties respectively. We assume that the deadline duration for interval $INT1$ is one unit of time and the delay duration for $INT2$ is two units of time. Timing interval $INT1$ is

triggered by event $e1$ and responded to by event $e2$. Interval $INT2$ is triggered upon model initialisation or by trigger-response event $e2$.

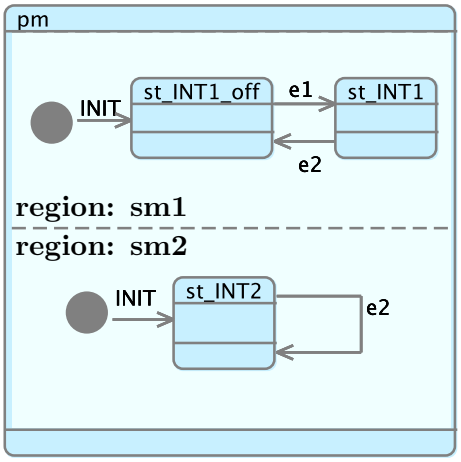

Figure 7.14: Parallel refinement representation in a state machine
$INT1(X;\ e1;\ e2;\ --;\ TP_1(DDL, C(1)))$
$INT2(X \times \{0,1\}, INIT, e2;\ e2;\ --;\ TP_1(DLY, C(2)))$

Figure 7.15 illustrates the dynamics of the given example model. We investigate only a subset of the whole execution tree. Hence, while in a certain state only some of the events that serve the example purposes are displayed as enabled or blocked.

The model starts in state $s1$ with a triggered interval $INT2$ instance $(X_1 \mapsto 0)$ and its clock set to zero. Interval $INT1$ is not active. We abbreviate clock variable names as $INT1\_$ and $INT2\_$ and omit other timing interval related variables. The figure illustrates only two of all the possible execution paths: one path demonstrates a successful response to both intervals, the other demonstrates the occurrence of the deadlock.

In the first execution path, the time progresses by one unit of time to state $s2$. Response event $e2$ is blocked by the active $INT2$ instance due to the unmet delay timing property requirement – $INT2\_clocks(X \mapsto 0)$ has not yet reached the minimum delay duration of two time units (blocking guard "g1" in the figure). Event $e1$ triggers the interval $INT1$ instance X_1 and the model enters state $s3$. The time progresses further by one time unit ($s4$). The *tick* event is now blocked because the $INT1$ instance has reached its deadline duration limit of one time unit (blocking guard "g1" in the figure). Response event $e2$ becomes enabled as $INT1$ has reached the minimum required delay duration of two time units. Both interval instances are responded to by executing event $e2$ ($s5$).

In the second execution path, event $e1$ triggers the $INT1$ instance and enters state $s6$. Analogous to state $s2$, the response event $e2$ is blocked by guard $g1$. Event *tick* progresses the time and the model enters state $s7$. Event *tick* is blocked by the $INT1$ deadline guard $g2$ – the time cannot progress beyond one time unit. Response event $e2$

is blocked by guard $g1$ until the time progresses to two time units. Since both events are blocked, we have a deadlock.

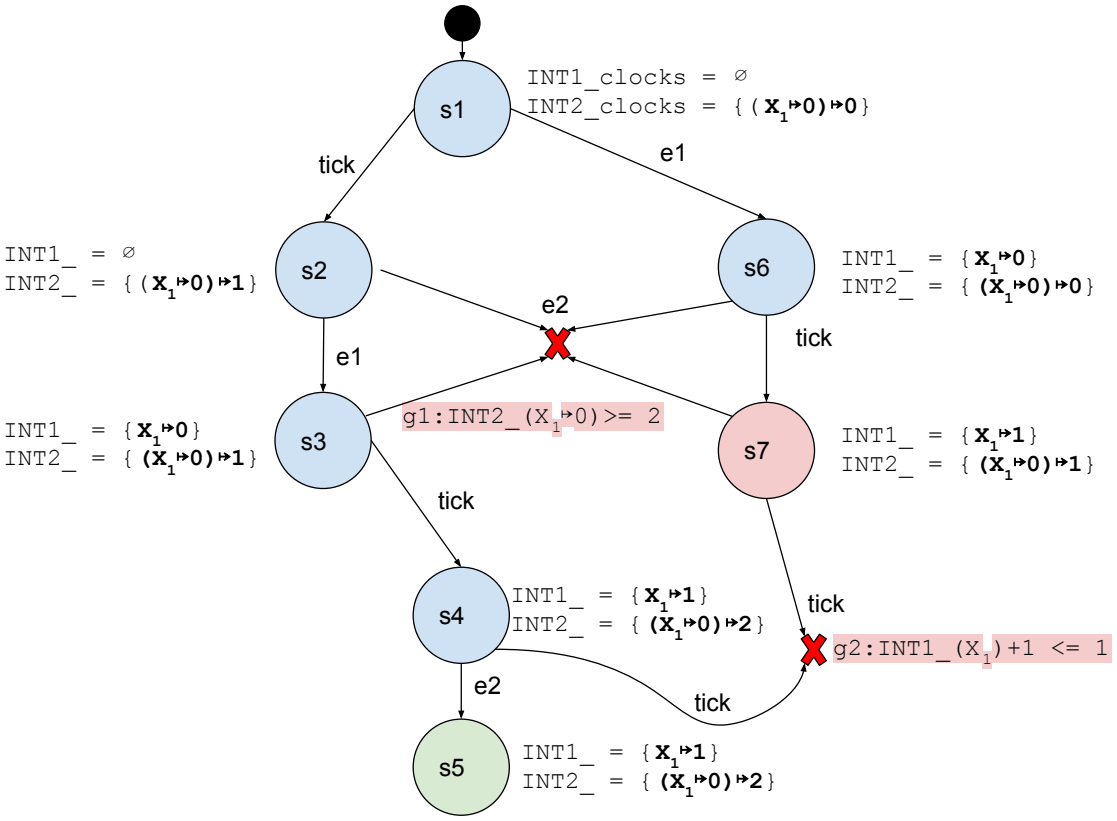

Figure 7.15: Parallel refinement

Natively, Event-B can verify for the absence of deadlocks in a machine by proving that at any point in time at least one event is enabled. Formally we express this as a proof obligation (7.21), with axioms A and invariant J in the hypothesis and a disjunction of all event guard blocks $G_1, ..., G_n$ as a goal:

$$\begin{array}{l} A \\ J \\ \vdash \\ G_1 \vee ... \vee G_n \end{array} \tag{7.21}$$

This proof obligation is not automated by the Rodin tool set and has to be constructed manually. Typically, such a proof obligation can become very complex, difficult to prove and to debug. For these reasons deadlock verification by proof in Event-B does not scale well and is considered impractical [163].

A more effective approach to detect deadlocks is model-checking. The Rodin platform uses ProB finite-state model-checker for this purpose. We used ProB to model-checker for our cases studies (chapters 9, 10 and 11) and found them deadlock-free.

7.5 Overview and Conclusions

We have provided guidelines on how timing interval and its refinement transformations can be graphically represented with state machine diagrams. We provide a method for synchronising timing interval and state-machines instances.

The guidelines for integrating iUML with the timing interval are based on our modelling experience. We have demonstrated that our approach maps well to iUML visual and generated Event-B constructs. Good mapping results can be attributed to the fact that timing interval, like iUML, is based on the superposition refinement. Hence the similar modelling practices.

Sarshogh's parametrised approach, discussed in subsection 3.6.1, could be mapped to multi-instance iUML diagrams with adjustments in the modelling style. First, parallel refinement, such as explained in subsection 5.1.6, is possible only in adjacent refinements. Sarshogh's data refines abstract variables, it is therefore not possible to refer to them from later refinements. Second, Sarshogh's timing properties support only one trigger event that can be further refined into multiple events. Similarly should be restricted to iUML state machine modelling style. Finally, the lack of abort event prevents mapping to the iUML "Any" pseudo-state.

The given guidelines have been successfully applied in three case studies: the pacemaker (chapter 9), the message passing protocol (chapter 10) and the landing gear system (chapter 11).

Chapter 8

Case Studies

To validate the timing interval and its refinement transformations we have performed three case studies: the heart pacemaker (chapter 9), the message passing protocol (chapter 10) and the landing gear system (chapter 11). Each case study validates a subset of the features in our approach. We assume that the timing units in all case studies are in milliseconds. We did not use expiry in any of the case studies.

The heart pacemaker case study demonstrates how a single-instance timing interval can be gradually elaborated via refinement chain to multiple concrete intervals. The timing intervals use combinations of delay and deadline timing properties with constant and dynamic durations. We formally synchronise iUML state machine constructs and cyclic timing intervals.

The message transfer protocol case study presents multiple timing interval instances. The functional part of the protocol is modelled with multi-instance iUML state machines and synchronised with the timing interval. The case study demonstrates the use of more complex Single-to-Multi and Retry transformations. We show how to manually synchronise two multi-instance timing intervals by their indices and events, and how to identify future work for automation.

The landing gear system case study demonstrates the multi-instance aspect of the timing interval. The system runs two identical controllers in parallel, hence two identical sets of timing intervals. We show that these timing intervals do not interfere.

8.1 Prover Optimisations

Constructing timing interval semantics from templates produces a finite set of possible Event-B constructs. As a result, the timing interval always generates a deterministic set of proof obligations. This allowed us to examine the generated POs and identify the

prover tactics required to discharge them. We then used this knowledge to optimise the auto-prover profile to tackle timing-interval-related proof obligations.

The three case studies together generated 6403 proof obligations of which the fine-tuned prover was able to discharge all but two, whereas the default auto-prover discharged 27%. Table 8.1 shows a per case study comparison of the proof obligation discharge rate between the fine-tuned and the default auto-provers. Numbers in brackets indicate the number of automatically discharged proof obligations. The fine-tuned auto-prover discharged all timing interval related POs. The two POs in the message passing protocol case study relate to the functional part and had to be proven manually.

Case Study	POs	Fine-Tuned Prover	Default Prover
Pacemaker	1551	100%	44% (690)
Msg. Passing	1015	99.9% (1013)	32% (318)
Landing Gear	3837	100%	19% (723)
Total	**6403**	**99.9% (6401)**	**27% (1731)**

Table 8.1: Comparison of fine-tuned and default auto-provers.

The optimised auto-prover was sufficient to discharge all timing interval related proof obligations and most of the functional-aspect-related proof obligations. This can be explained by the fact that our models are focused on the timing requirements, and that the functional part of the models is relatively simple.

We overview the structure of the auto-prover profile. The exported auto-prover profile can be found in (Appendix B). We wrap the list of proof tactics in a "Loop on All Pending" combinator which applies tactics in the given order until one succeeds, it then restarts from the first tactic on the next pending node in the PO tree. The tactics list can be split into three parts. The first part of the list comprises tactics that simplify hypotheses and the goal[1]:

- "Partition Rewriter" - expands predicates "partition(...)" in visible hypotheses and goal.
- "Simplification Rewriter" - tries to simplify all predicates in a sequent using pre-defined simplification rewriting rules.
- "For-all rewriter" - simplifies any sequent with a universally-quantified goal by freeing all its bound variables.
- "Implicative Goal" - simplifies any sequent with an implicative goal by adding the left-hand side of the implication to the hypotheses and making its right-hand side the new goal.

[1]Tactics' descriptions taken from http://wiki.event-b.org/index.php/Rodin_Proof_Tactics

- "Remove all equivalences in goal" - rewrites all equivalences into implications conjunction in goal.
- "Remove all Membership/Inclusion in goal" - removes all the memberships and inclusions in the goal.

The order of the tactics is the same as required to discharge a timing interval proof obligation manually. We omitted the rest of the default tactics because oversimplifying the hypotheses and the goal sometimes hinders the ability of SMT provers to discharge the proof obligations.

In the second part of the tactics list, we first apply lasso to shortlist only the relevant hypotheses. Then we attempt to discharge the proof obligation with the SMT solvers: CVC3, CVC4, veriT[2] and Z3[3]. We set SMT solver timeout to 5 seconds. Most of the automatic and speedy proof discharge can be attributed to SMT solvers.

In case the SMT solvers fail, we attempt to discharge the proof obligation with AtelierB Mono-Lemma and Meta provers. From our experience, Mono-Lemma prover was faster and more successful in discharging the proof obligations than the Predicate prover. Relevance Filter (Meta prover) has a high PO discharge rate but takes a considerable amount of time. We therefore placed it last.

[2]veriT version veriT-stable2016
[3]Z3 version 4.5.1

Chapter 9

Pacemaker Case Study

In this chapter, we give details on the pacemaker case study. Firstly, we briefly overview the pacemaker system. Then, give detail on the modelling process. Finally, we discuss the resulting formal model of the pacemaker system regarding timing interval applicability, its validation and verification.

9.1 Pacemaker

The artificial pacemaker is a medical device that uses electrical impulses, delivered and sensed by electrodes contracting the heart muscles, to regulate the beating of the heart. The primary purpose of a pacemaker is to maintain an adequate heart rate, either because the heart's natural pacemaker is not fast enough, or there is a block in the heart's electrical conduction system.

In this work we have modelled a two-lead device (Figure 9.1), meaning that one lead paces and senses the right atrium, and the other lead paces and senses the right ventricle. Typically, the pacemaker's type and function are defined by a five-letter coding system. In this work, a simplified three-letter DDD pacemaker has been used. The code DDD indicates that the pacemaker is capable of pacing and sensing atrium and ventricle chambers as described further.

The pacemaker's functionality depends on its internal model of a normal heart, displayed in the example ECG schematic in Figure 9.2. The normal heart is modelled regarding a set of interdependent nested cyclic timing intervals, representing various requirements of the normal pacing cycle. The model describes electrical activity in each of the two pacemaker channels, representing the atrial and ventricular heart chambers respectively. A channel may *sense* an intrinsic electrical signal from the heart, resulting in the contraction of its chambers. A channel may be subjected to a *pace*-actuating signal from

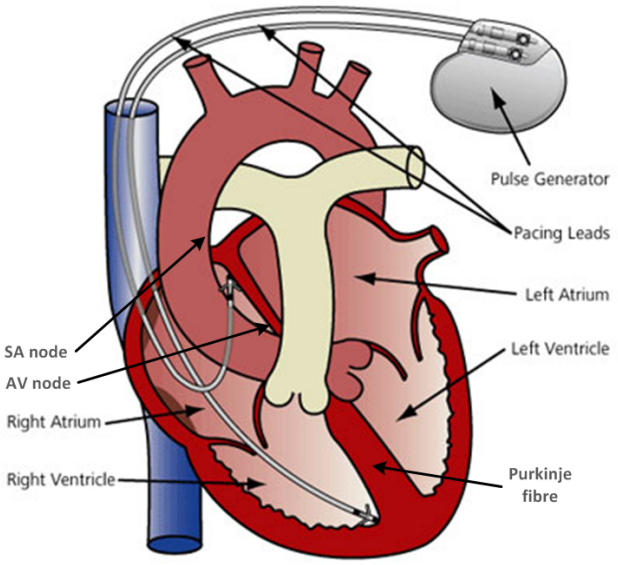

Figure 9.1: Heart and pacemaker [1]

the pacemaker to initiate contraction, in the absence of a timely sensed signal. These and similar events define the bounds of the modelled timing intervals.

We use a simplified definition of the *cardiac cycle* to define a period between two ventricular contractions (Figure 9.2). The cycle is initiated by any ventricular activity and triggers a set of post-ventricular activity timing intervals $V_0, \ldots V_k$. Typically atrium activity follows, triggering a set of atrial timing intervals $A_0, \ldots A_n$ that denote post-atrium events. The number of intervals (n and k) can vary, so as their duration. Cycles are distinguished by j in the figure to mark their order.

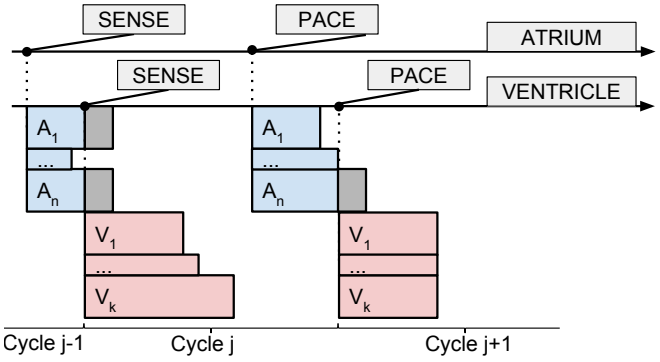

Figure 9.2: Example of the schematic ECG

We give details on the requirements during our narration of the pacemaker-modelling process. Various sources of cardiology [35, 92, 62, 152, 150, 59, 26] were analysed in order to produce an unambiguous-requirements document (section D.1). A total of 48 requirements were identified for our DDD-type pacemaker cycle. They are classified into

[1] Image taken from the London Arythmia Centre webpage, http://www.londonarrhythmiacentre.co.uk /treatments-pacemakers.html

environment assumptions (EVN), functional (FUN) and timing (TIME). Our scope was limited to time interval modelling. The hardware aspect was therefore abstracted away in this work. Faults were not considered in this work, based on the assumption that the hardware would not fail.

The heart's normal activity, including four abnormal heart behaviour patterns, was considered as an expected behaviour of that environment. The abnormal behaviour patterns include: 1. absence of atrial sense; 2. absence of ventricular sense; 3. premature atrial contraction, during which the atrial event is not intervened (preceded) by a ventricular event; and 4. premature ventricular contraction that is not intervened by an atrial event.

9.2 Modelling the Pacemaker

In this case study we demonstrate how a timing interval can be elaborated through multiple levels of refinement. Specifically, we make use of the Alternative (subsection 5.1.1), Sub-Interval (subsection 5.1.2) and Abort-to-Response (subsection 5.1.3) refinement transformations. The timing intervals use combinations of delay and deadline timing properties with constant and dynamic durations. We used only delay and deadline timing properties in the case study.

The pacemaker system is modelled following the guidelines described in chapter 7. We construct the functional part of the pacemaker model using iUML state machine diagrams. Timing intervals are then superimposed using the tiGen tool. We formally integrate variable-based iUML state machine constructs and the timing intervals from tiGen. The timing interval and its refinements are visually represented as per convention, given in the guidelines chapter. Most of the model has been generated automatically. We have manually added the following Event-B elements: 1. invariants to synchronise state machine state with timing interval; 2. dynamic duration gluing invariants for refinement transformations. We note parts of the model that have been modelled manually in figures by starting a caption with *MANUALLY added:*. Automatically generated code, displayed in figures, starts with *AUTO generated:*. The pacemaker model consists of six refinement levels.

9.2.1 Abstract Machine: Cardiac Cycle

In the abstract machine we define a combination of *upper rate interval* (URI) and *lower rate interval* (LRI). Interval *URI* defines the minimum duration URI_t_{DLY} (TIME - 24) that must pass from the last ventricular pace or sense event (TIME - 29) until the pacemaker is able to pace the ventricles. Triggered by ventricular pace or sense event (TIME - 4), interval *LRI* defines the longest allowed duration LRI_t_{DDL} (TIME - 1)

without ventricular sense, at the end of which the pacemaker must deliver the ventricular pace (TIME - 2). In other words, interval *LRI* defines the longest possible cardiac cycle.

Consider an example electro-cardiogram schematic (Figure 9.3) illustrating the dynamics of the aforementioned intervals in three cardiac cycles.

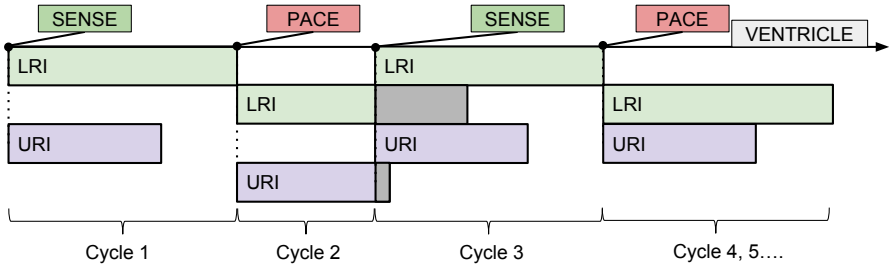

Figure 9.3: *URI_LRI* interval.

The first cardiac cycle, initiated by the ventricular sense, triggers the *LRI* and *URI* intervals. No ventricular sense is detected throughout the cycle; thus, a ventricular pace is delivered at the end of *LRI* thereby ending the first cycle and starting the second one. During the second cycle, a heart contraction is sensed while the *URI* interval is active. The event resets both intervals and starts the third cardiac cycle. Note that while *URI* prevents the pacemaker's pace, the sense event is not constrained. The third cycle ends with the pacemaker delivering the pace due to the absence of sense event in the heart.

The combination of the two intervals – we define it as an *URI_LRI* timing interval – represents the cardiac cycle with restrictions on the ventricular pace. The pacemaker is permitted to intervene with the ventricular pace no earlier than URI_t_{DLY} after the cardiac cycle is started. In case the cardiac cycle is not restarted by ventricular sense, the ventricular pace must be delivered by the end of the cardiac cycle duration LRI_t_{DDL}. We represent the interval in the iUML state machine diagram as state *URI_LRI* (Figure 9.4). Events *vp* and *vs* represent ventricular pace and sense respectively (ENV-1).

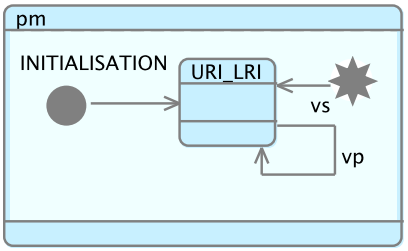

Figure 9.4: Abstract machine.: *URI_LRI* interval.

$$URI\_LRI(X_s;\ INIT, vp, vs;\ vp;\ vs;\\ TP_1(DLY, C(URI\_t_{DLY})), TP_2(DDL, C(LRI\_t_{DDL}))) \tag{9.1}$$

We specify interval *URI_LRI* in our timing interval specification (9.1). The interval has three trigger events: *vs*, *vp* and *INIT*. The latter means that the interval is active

upon model initialisation. At any point, the cycle can be aborted by *vs*. Event *vp* is permitted only after time URI_t_{DLY} has passed and must fire at the end of LRI_t_{DDL}.

We consider only single instance intervals with index set X, meaning that at the most only one interval *URI_LRI* instance can be running at any point in time. The subscript X_s in the set specification does not mean that the cardinality set X is specified. The indicator instructs the tiGen tool to generate an invariant (Figure 9.5) to enforce the singleton timing interval. The singleton timing interval is presented in section 4.7. Variables *URI_LRI_trig*, *URI_LRI_resp* and *URI_LRI_abrt* store the interval's trigger, response and abort indices respectively.

$$\text{inv1}: \ \text{URI\_LRI\_trig} \setminus (\text{URI\_LRI\_resp} \cup \text{URI\_LRI\_abrt}) \neq \varnothing \Rightarrow (\exists\, \text{idx} \cdot \text{idx} \in \text{X} \wedge \text{URI\_LRI\_trig} \setminus (\text{URI\_LRI\_resp} \cup \text{URI\_LRI\_abrt}) = \{\text{idx}\})$$

Figure 9.5: AUTO generated: singleton invariant for interval *URI_LRI*.

We ensure the relation between state *URI_LRI* and interval *URI_LRI* with invariant (Figure 9.6). The invariant states that the interval is always active when state *URI_LRI* is active and vice versa. This invariant has been written manually because integration of the iUML and tiGen tool is not implemented yet. The initialised state *URI_LRI* is always active (Figure 9.4), as is the timing interval.

$$\text{inv1}: \ \text{URI\_LRI} = \text{TRUE} \Leftrightarrow (\exists\, \text{idx} \cdot \text{idx} \in \text{X} \wedge \text{URI\_LRI\_trig} \setminus (\text{URI\_LRI\_resp} \cup \text{URI\_LRI\_abrt}) = \{\text{idx}\}$$

Figure 9.6: MANUALLY added: sync. invariant for interval *URI_LRI* and the corresponding sate.

The principle of ensuring the at most one active single instance and specifying the relation between corresponding states and timing intervals is analogous and applied for all intervals discussed in this case study.

9.2.2 First Refinement: Post- Ventricular and Atrial Intervals

The cardiac cycle, as defined in the previous section, can be split into a sub-interval sequence of *ventriculoatrial interval* (*VAI*) and *atrioventricular interval* (*AVI*). Interval *VAI* takes the first part of the abstract interval and represents the post-ventricular period, whereas interval *AVI* takes the second half and represents the post-atrial period. The concrete intervals do not overlap (TIME - 33). We model sub-intervals as separate states *VAI* and *AVI* (Figure 9.7). Events *as* and *ap* represent atrial sense and pace activities respectively. We visualise the dynamics of the two intervals in Figure 9.9 later.

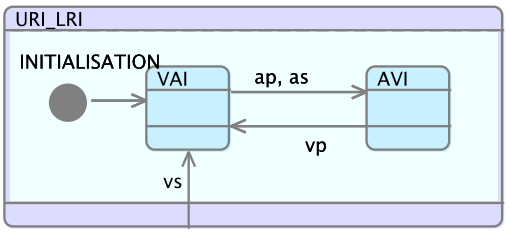

Figure 9.7: 1st ref.: Sub-intervals *VAI* and *AVI*.

$$VAI(X_s;\ INIT, vp, vs;\ ap;\ as, vs;\\ TP_1(DLY, C(VAI\_t_{DLY})), TP_2(DDL, C(VAI\_t_{DDL})) \tag{9.2}$$

$$AVI(X_s;\ ap, as;\ vp;\ vs;\\ TP_1(DLY, D(AVI\_t_{DLY})), TP_2(DDL, D(AVI\_t_{DDL})) \tag{9.3}$$

We refine abstract timing interval *URI_LRI* by applying sub-interval refinement transformation (subsection 5.1.2) to two sub-intervals. The *VAI* sub-interval (9.2) shares the same trigger events as the abstract interval (Figure 9.8) – it is triggered upon model initialisation or by any ventricular event (TIME - 29). At any point in time it can be aborted by *vs* or *as* events (TIME - 32). If not aborted, *ap* is delivered after exactly *VAI_t* time units have passed (TIME - 31).

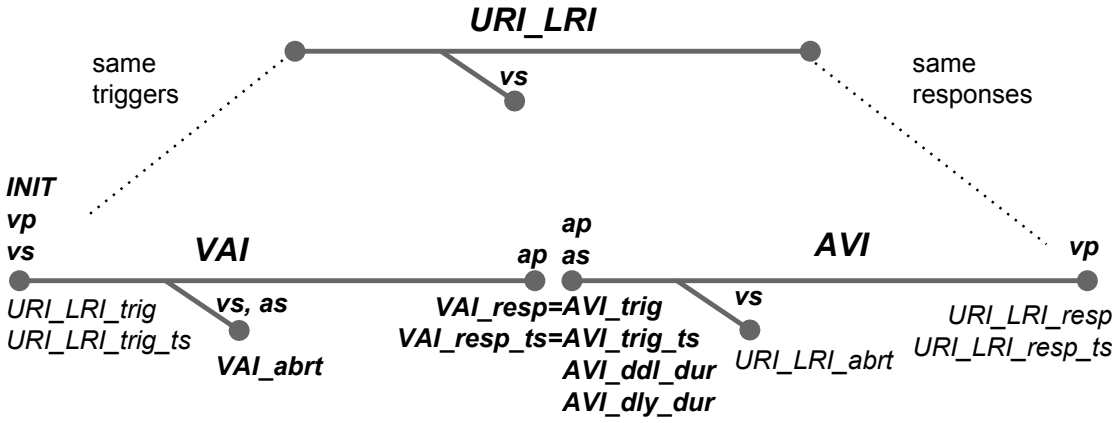

Figure 9.8: 1st ref.: Sub-intervals *VAI* and *AVI* represented in iUML state machines.

Interval *VAI* duration is derived from *LRI* and *AVI* interval durations. Hence $VAI\_t = LRI\_t_{DDL} - AVI\_t_{min}$, where AVI_t_{min} is the shortest specified interval *AVI* duration (TIME - 28, TIME - 30, TIME - 14, TIME - 15). Unless stated otherwise, intervals discussed in this chapter have the same delay and deadline timing property durations. Such durations are written without a timing property type, e.g. VAI_t and AVI_t for *VAI* and *AVI* intervals respectively. In the specification we still use timing property identifiers to indicate which timing properties are being used, e.g. $VAI\_t_{DLY,DDL}$ for *VAI* in (Figure 9.8), denotes that the interval has delay and deadline timing properties.

Interval *AVI* (9.3) follows the *VAI* interval and is triggered by either one of the shared events *ap* or *as* (TIME - 10). Event *ap* serves as a response for *VAI* and as a trigger for *AVI*. Event *as* serves as an abort for the *VAI* interval and as a trigger for the *AVI* interval. We remind the reader that the abort-trigger event, such as *as*, can be introduced for adjacent sub-intervals as explained in subsubsection 5.1.2.2. While *AVI* is active, atrial sense is ignored (TIME - 12). Interval *AVI* is a single-instance timing interval. Atrial events *ap* and *as*, which serve as the triggers, will therefore remain blocked until the interval is responded to or aborted. Moreover, we use iUML state machine behaviour to implement this requirement: transition *as* is enabled only when its source state *VAI* is active, and blocked when state *AVI* is active.

Interval *AVI* can be aborted (TIME - 13) by *vs*, otherwise it must be responded to by *vp* at the end of the interval (TIME - 11).

Interval *AVI* has a dynamic timing property duration to model a Wenckebach upper-rate response. To explain the Wenckebach effect, consider the first three cardiac cycles in the example scenario in Figure 9.9. The first cycle is initiated by *vs*. While *VAI* is active, no atrium sense is detected. The interval is therefore responded to with *ap*. Ventricular sense does not occur, therefore at the end of *AVI* the ventricular pace is delivered and the second cardiac cycle is started. This time, interval *VAI* is aborted by an atrial sense. Again, *vp* is delivered at the end of the *AVI* interval. Note that in both cycles *vp* is delivered after duration URI_t_{DLY} has passed from the last ventricular event occurrence. In the third cycle, *as* occurs early enough so that the deadline duration of interval *AVI* arrives sooner than the delay of *URI* expires. This means that *AVI* cannot fulfil its requirement to deliver the *vp* at the end of the interval. To address such a scenario, we extend the *AVI* duration to match the end of the *URI* delay duration. The length of the extension can differ from cycle to cycle. Therefore, we use dynamic duration to record the duration of each cycle. Such an *AVI* interval duration extension is called Wenckebach upper-rate response.

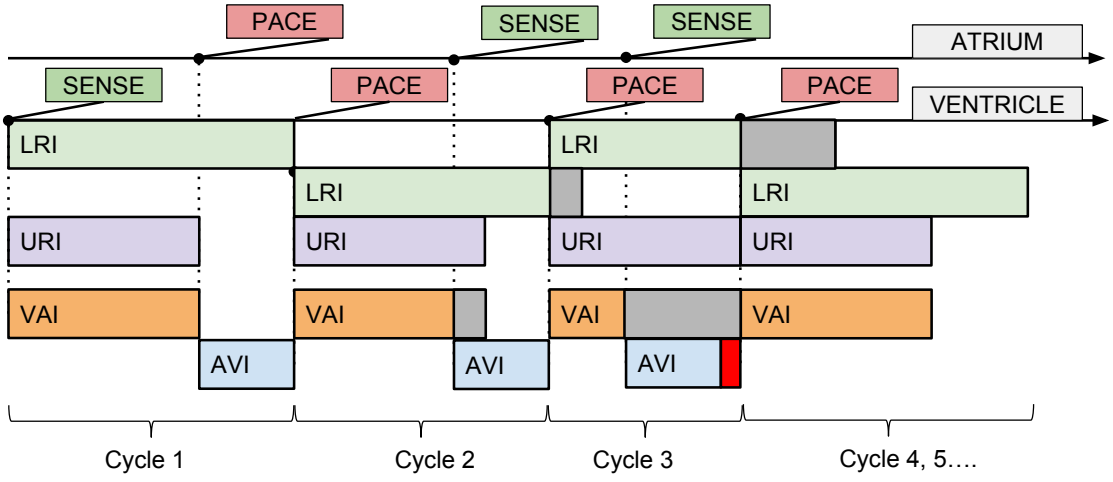

Figure 9.9: Example ECG with Wenckebach effect.

Before explaining the semantics of the *AVI* dynamic duration, we display the variables associated with each interval after the sub-interval transformation (Figure 9.8). Interval *VAI* uses variable *VAI_abrt* to store abort indices and *VAI_resp* and *VAI_resp_ts* variables to store the response-related information. Interval *AVI* uses a set of new trigger-related variables and two variables to store dynamic delay and deadline durations – *AVI_dly_dur* and *AVI_ddl_dur*.

The duration of the interval is calculated in the trigger events *as* and *ap* (Figure 9.10). Parameter *p_URI_LRI_elapsed* takes the elapsed time of interval *URI_LRI* from the corresponding clock variable *URI_LRI_clocks* (*grd*1). We then consider two cases. In the first case (*grd*2), when the pre-set minimum duration AVI_t_{min} is shorter than the remaining time until interval *URI_LRI* delay is satisfied, *AVI* duration is assigned that duration. In the second case (*grd*3), if the remaining time is less than the minimum *AVI* duration, the *AVI* duration is set to the pre-reset minimum value of AVI_t_{min}. Guard *grd*4 ensures the *AVI* delay and deadline duration are equal.

$$
\begin{array}{ll}
\text{grd1}: & \text{p\_URI\_LRI\_elapsed} \in \text{ran}((\text{URI\_LRI\_resp} \cup \text{URI\_LRI\_abrt}) \lhd\!\!\!\!- \text{URI\_LRI\_clocks}) \\
\text{grd2}: & \text{URI\_t}_{\text{DLY}} - \text{p\_URI\_LRI\_elapsed} > \text{AVI\_t}_{\text{min}} \Rightarrow \text{p\_AVI\_dly\_val} = \text{URI\_t}_{\text{DLY}} - \text{p\_URI\_LRI\_elapsed} \\
\text{grd3}: & \text{URI\_t}_{\text{DLY}} - \text{p\_URI\_LRI\_elapsed} \leq \text{AVI\_t}_{\text{min}} \Rightarrow \text{p\_AVI\_dly\_val} = \text{AVI\_t}_{\text{min}} \\
\text{grd4}: & \text{p\_AVI\_dly\_val} = \text{p\_AVI\_ddl\_val}
\end{array}
$$

Figure 9.10: MANUALLY added: interval *AVI* duration determination condition

Consistency between the abstract interval and the concrete durations is ensured by two invariants (Figure 9.11). Invariant *inv*1 ensures the *AVI* delay duration is equal to or greater than the abstract duration URI_t_{DLY}. The invariant does not use a sum of *VAI* and *AVI* delay durations due to the *as* event serving as the abort-trigger event. This event can abort *VAI* at any point in time. The rule is explained in subsubsection 5.1.2.2. Invariant *inv*2 specifies the relationship between abstract and concrete deadline durations in an analogous manner.

$$
\begin{array}{ll}
\text{inv1}: & \forall \text{idx} \cdot \text{idx} \in \text{AVI\_trig} \Rightarrow \text{URI\_LRI\_trig}_{\text{t}}\text{s}(\text{idx}) + \text{URI\_t}_{\text{DLY}} \leq \text{AVI\_trig\_ts}(\text{idx}) + \text{AVI\_dly\_dur}(\text{idx}) \\
\text{inv2}: & \forall \text{idx} \cdot \text{idx} \in \text{AVI\_trig} \Rightarrow \text{LRI\_t}_{\text{DDL}} \geq \text{AVI\_trig\_ts}(\text{idx}) + \text{AVI\_ddl\_dur}(\text{idx})
\end{array}
$$

Figure 9.11: MANUALLY added: gluing invariants for delay and deadline duration consistency between *URI_LRI* and the sequence of *VAI* and *AVI* sub-intervals.

9.2.3 Second Refinement: Post-Ventricular Refractory Intervals

The initial portion of the *VAI* interval consists of the *ventricular refractory period* (*VRP*) and *post-ventricular atrial refractory period* (*PVARP*), during which the pacemaker

cannot sense any signals. The intervals partially overlap and are active in parallel. More specifically, during *VRP* the pacemaker ignores signals in the ventricular channel for the duration of *VRP_t* (TIME - 23) and during *PVARP* – signals in the atrium channel for the duration of *PVARP_t* (TIME - 21).

We elaborate interval *VAI* with a Sub-Interval transformation (subsection 5.1.2) to a sequence of *VRP* and *VRP_off* (Figure 9.12). The first sub-interval denotes the refractory period being active, and the second denotes when it has ended but the *VAI* interval is still active.

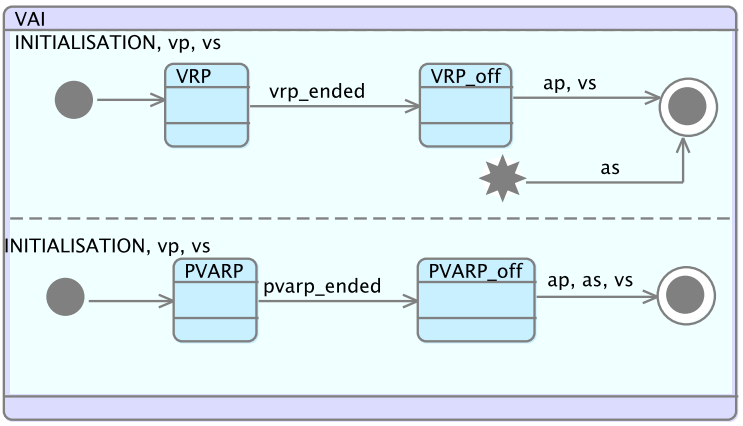

Figure 9.12: 2nd ref.: Two parallel sub-intervals, elaborating *VAI*. Upper region: *VRP* and *VRP_off*. Lower region: *PVARP* and *PVARP_off*.

$$VRP(X_s;\ INIT, vp, vs;\ vrp\_ended;\ as;\ TP_1(DLY, C(VRP\_t_{DLY})), TP_2(DDL, C(VRP\_t_{DDL})) \tag{9.4}$$

$$VRP\_off(X_s;\ vrp\_ended;\ ap;\ as, vs;\ TP_1(DLY, C(VRP\_off\_t_{DLY})), TP_2(DDL, C(VRP\_off\_t_{DDL})) \tag{9.5}$$

$$PVARP(X_s;\ INIT, vp, vs;\ pvarp\_ended;\ vs;\ TP_1(DLY, C(PVARP\_t_{DLY})), TP_2(DDL, C(PVARP\_t_{DDL})) \tag{9.6}$$

$$PVARP\_off(X_s;\ pvarp\_ended;\ ap;\ as, vs;\ TP_1(DLY, C(PVARP\_off\_t_{DLY})), TP_2(DDL, C(PVARP\_off\_t_{DDL})) \tag{9.7}$$

The *VRP* interval (9.4) shares trigger events with the abstract interval *VAI* (TIME - 6). Since during *VRP* the pacemaker ignores signals in the ventricular channel, abort event *vs* is removed from the specification (TIME - 5). Event *vs* is present in the abstract interval's specification but absent in the concrete one (Figure 9.13), is added a generated guard to prevent event's execution while *VRP* is active . The interval can still be aborted by *as* (TIME - 7). After time *VRP_t* has passed of triggering the interval, event *vrp_ended* ends interval *VRP* and starts *VRP_off*. Interval *VRP_off* (9.5) shares response and abort events with the abstract interval *VAI* (9.2). Note the ambiguity in

the diagram. Although events *ap* and *vs* serve different roles for interval *VRP_off*, in the diagram they are represented in the same way.

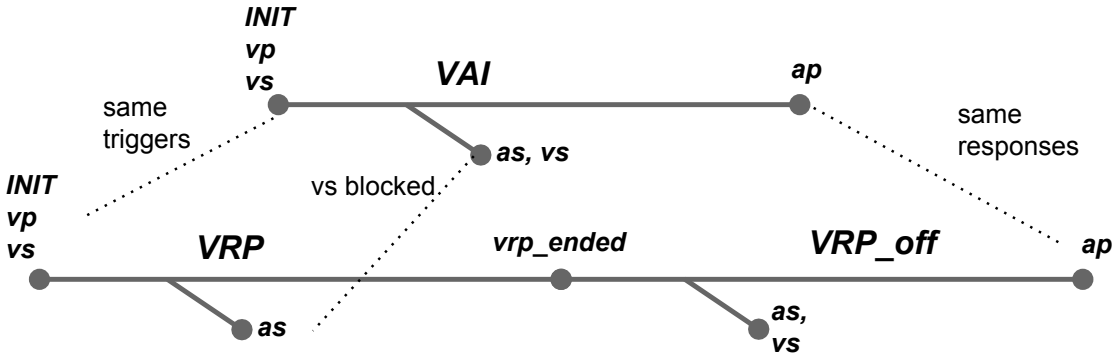

Figure 9.13: 2nd ref.: Sequence of *VRP* and *VRP_off* sub-intervals.

Axioms *axm*1 and *axm*2 in Figure 9.14 ensure the consistency of delay and deadline timing properties between the abstract interval *AVI* and the concrete sub-interval sequence of *VRP* and *VRP_off* intervals.

$$\text{axm1}: \ \texttt{VRP\_t} + \texttt{VRP\_off\_t} \leq \texttt{AVI\_t}$$
$$\text{axm2}: \ \texttt{VRP\_t} + \texttt{VRP\_off\_t} \geq \texttt{AVI\_t}$$

Figure 9.14: AUTO generated: gluing invariants for delay and deadline duration consistency between *URI_LRI* and the sequence of *VAI* and *AVI* sub-intervals.

We apply Sub-Interval refinement transformation again to transform the abstract interval *VAI* into the sequence of sub-intervals *PVARP* (9.6) and *PVARP_off* (9.7) in parallel to *VRP*[2]. The process is analogous to that of *VRP*. The sequence is triggered (TIME - 19) and responded to by the same events as interval *VRP* and the *VRP_off* sub-interval sequence. Event *as* is blocked while interval *PVARP* is active (TIME - 18), but the interval can still be aborted by *vs* (TIME - 20).

Duration consistency between concrete and abstract layers is expressed in axioms, which are analogous to *axm*1 and *axm*2 in Figure 9.14.

9.2.4 Third Refinement: Sensed and Paced AVI

The third refinement adds a finer-grained view of the *AVI* interval dynamics. In particular, we distinguish interval *AVI* as either initiated by the actuated atrial pace event or as the sensed intrinsic atrial event – intervals *pAVI* and *sAVI* respectively. We use the Alternative refinement transformation (subsection 5.1.1) to differentiate between the dynamics of the intervals are different, and therefore are their parameters. In this model, refined intervals *pAVI* and *sAVI* are treated differently in terms of their duration

[2]The parallelisation process is described in subsection 5.1.6)

constraints [35] (TIME - 17, TIME - 16). We model the intervals as two sub-states of state *AVI* in Figure 9.15.

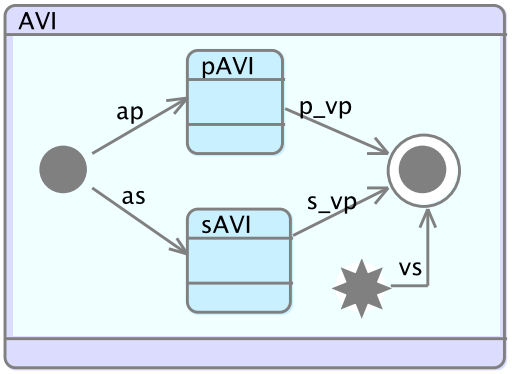

Figure 9.15: 3rd ref.: Alternative intervals *pAVI* and *sAVI*.

$$pAVI(X_s;\ ap;\ p\_vp;\ vs;\\ TP_1(DLY, D(pAVI\_t_{DLY})), TP_2(DDL, D(pAVI\_t_{DDL})) \tag{9.8}$$

$$sAVI(X_s;\ as;\ s\_vp;\ vs;\\ TP_1(DLY, D(sAVI\_t_{DLY})), TP_2(DDL, D(sAVI\_t_{DDL})) \tag{9.9}$$

Interval *pAVI* (9.8) is triggered by the paced atrial stimulus, represented by event *ap*, whereas interval *sAVI* (9.9) is triggered by the atrial sense event, represented by event *as*. Both events *ap* and *as* refine their abstract counterparts. Intervals *pAVI* and *sAVI* can be either aborted by *vs* or responded to by *p_vp* and *s_vp* respectively. Both response events refine event *vp*.

Both intervals have a dynamic duration to retain the Wenckebach upper-rate response introduced in subsection 9.2.2. Consistency of the timing property durations between intervals *AVI* and *pAVI* is ensured with two invariants (Figure 9.16), and similarly for *sAVI*. Invariants *inv*1 and *inv*2 ensure delay and deadline duration consistency respectively.

$$\begin{aligned} &inv1: \forall idx \cdot idx \in pAVI\_trig \Rightarrow AVI\_dly\_dur(idx) \leq pAVI\_dly\_dur(idx) \\ &inv2: \forall idx \cdot idx \in pAVI\_trig \Rightarrow AVI\_ddl\_dur(idx) \geq pAVI\_ddl\_dur(idx) \end{aligned}$$

Figure 9.16: MANUALLY added: gluing invariants for delay and deadline duration consistency between *AVI* and *pAVI*

9.2.5 Fourth Refinement: Ventricular Safety Period

After the start of *pAVI*, there is a minimum period called the *Ventricular Safety Period* (*VSP*) which is strictly shorter than the *AVI* – during this period a paced ventricular event is prohibited [35]. We introduce the first of two refinement transformations to

implement this. Through refinement we divide interval *pAVI* into a sequence of sub-intervals *VSP* and *VSP_off* (Figure 9.17).

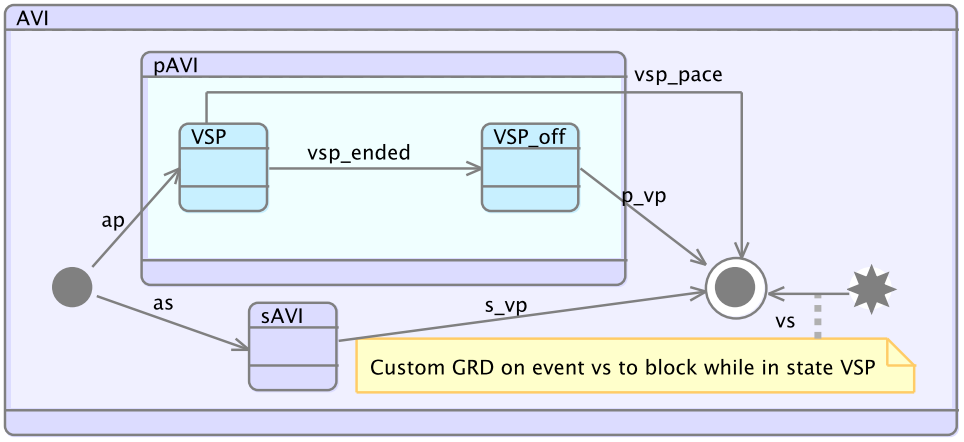

Figure 9.17: 4th ref.: *pAVI* refined to sub-intervals *VSP* and *VSP_off*.

$$VSP(X_s;\ ap;\ vsp\_ended;\ vsp\_pace;\ TP_1(DLY, C(VSP\_t_{DLY})), TP_2(DDL, C(VSP\_t_{DDL})) \tag{9.10}$$

$$VSP\_off(X_s;\ vsp\_ended;\ p\_vp;\ vs;\ TP_1(DLY, D(VSP\_off\_t_{DLY})), TP_2(DDL, D(VSP\_off\_t_{DDL})) \tag{9.11}$$

Sub-interval *VSP* (9.10) is triggered by event *ap* (TIME - 50). New event *vsp_ended*, refining *skip*, serves simultaneously as the response event for interval *VSP* and as the trigger event for interval *VSP_off* (9.11). Interval *VSP_off* is responded to by event *p_vp* after time *VSP_t* (TIME - 53). Interval *VSP* may be aborted by event *vsp_pace*, refining *vs*, whereas *VSP_off* may be aborted by *vs*.

At the functional level, we block event *vs* with a custom guard $VSP = false$. Because *vs* is removed from the concrete specification (9.10), the tiGen tool adds an additional guard to block the event while interval *VSP* is active (Figure 9.18). The guard says that the abort parameter must not have the index of an active *VSP* instance.

$$\texttt{grd1}: \ \texttt{p\_URI\_LRI\_abrt} \cap (\texttt{pAVI\_trig} \setminus \texttt{VSP\_resp}) = \varnothing$$

Figure 9.18: AUTO generated: gluing invariants for delay and deadline duration consistency between *AVI* and *pAVI*

The duration of interval *VSP* is fixed to *VSP_t* (TIME - 54). In order to retain the Wenckebach response, the *VSP_off* duration remains dynamic, with a minimum duration value VSP_off_min set to $AVI_{min} - VSP\_t$. The algorithm to calculate the actual duration for the interval instance is analogous to that of Figure 9.10.

Three gluing invariants ensure the consistency of delay and deadline durations (Figure 9.19). Invariant *int*1 tackles the delay case when *VSP* and *VSP_off* have been

responded to. The sum of both interval durations must be greater or equal to the abstract interval *pAVI* delay duration. Similarly, interval *int*2 deals with the deadline timing duration. Invariant *inv*3 deals with the case where *VSP_off* has not yet been activated and ensures that *VSP* does not exceed the abstract deadline duration.

$$
\begin{array}{l}
\texttt{inv1}: \forall\, \texttt{idx} \cdot \texttt{idx} \in \texttt{VSP\_off\_trig} \Rightarrow \texttt{VSP\_off\_dly\_dur(idx)} + \texttt{VSP\_t} \geq \texttt{pAVI\_dly\_dur(idx)} \\
\texttt{inv2}: \forall\, \texttt{idx} \cdot \texttt{idx} \in \texttt{VSP\_off\_trig} \wedge \texttt{idx} \in \texttt{pAVI\_trig} \Rightarrow \texttt{VSP\_off\_ddl\_dur(idx)} + \texttt{VSP\_t} \leq \texttt{pAVI\_ddl\_dur(idx)} \\
\texttt{inv3}: \forall\, \texttt{idx} \cdot \texttt{idx} \notin \texttt{VSP\_off\_trig} \wedge \texttt{idx} \in \texttt{pAVI\_trig} \Rightarrow \texttt{VSP\_t} \leq \texttt{pAVI\_ddl\_dur(idx)}
\end{array}
$$

Figure 9.19: MANUALLY added: gluing invariants for delay and deadline duration consistency between *pAVI* and *VSP* and *VSP_off* sub-intervals.

9.2.6 Fifth Refinement: Abort-To-Response Transformation

We use Abort-to-Response transformation (subsection 5.1.3) to complete the modelling of *VSP*. If either ventricular pace or sense occurs while the *VSP* interval is active, a ventricular pace must be delivered at the end of the interval and not before ([35])(TIME - 51). This is achieved by refining the abort event *vx* of interval *VSP* - which can fire at any time - into the response *vsp_pace*. This new event represents the ventricular pace at the end of the *VSP* sub-interval. Note that *vsp_pace* only works locally as the response event for interval *VSP_atr*. The event retains its role as the abort event for the sub-interval abstract sequence of *VSP* and *VSP_off*.

$$
\begin{array}{l}
VSP\_atr(X_s;\ ap;\ vsp\_pace, vsp\_ended;\ --; \\
\qquad TP_1(DLY, C(VSP\_atr\_t_{DLY})), TP_2(DDL, C(VSP\_atr\_t_{DDL})))
\end{array}
\tag{9.12}
$$

We add a finer-grained view of the *VRP* (Figure 9.20) and *PVARP* (Figure 9.21) interval dynamics. The process is analogous to the transformation in Figure 9.15. Interval *VRP* is refined to alternative intervals *pVRP* and *sVRP*. Interval *PVARP* – to *pPVARP* and *sPVARP*.

pVRP is then initiated by ventricular pace events *p_vp*, *vsp_pace* and *s_vp* (9.13). *sVRP* is triggered by the intrinsic *vs* (9.14). Event *p_vrp_ended* responds to *pVRP* and *s_vrp_ended* – to *sVRP*. Both response events refine event *vrp_ended*. The principle of *sPVARP* and *pPVARP* intervals is analogous to the *VRP* counterparts.

9.2.7 Sixth Refinement: Blanking Periods

Blanking periods help to avoid crosstalk when one chamber gets paced by the pacemaker and the signal is sensed in the other chamber. While in a refractory period, the

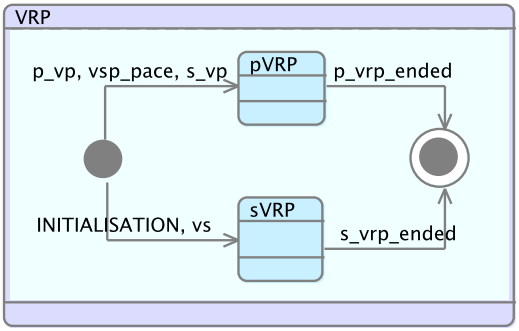

Figure 9.20: 5th ref.: alternative intervals $pVRP$ and $sVRP$.

$$pVRP(X_s;\ p\_vp, s\_vp, vsp\_pace;\ p\_vrp\_ended;\ as;\ TP_1(DLY, C(pVRP\_t_{DLY})), TP_2(DDL, C(pVRP\_t_{DDL}))) \tag{9.13}$$

$$sVRP(X_s;\ INIT, vs;\ as, s\_vrp\_ended;\ TP_1(DLY, C(sVRP\_t_{DLY})), TP_2(DDL, C(sVRP\_t_{DDL}))) \tag{9.14}$$

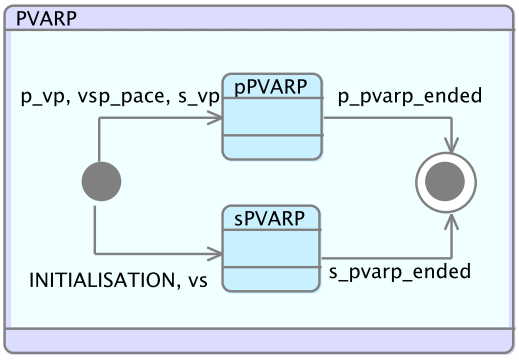

Figure 9.21: 5th ref.: alternative intervals $pPVARP$ and $sPVARP$.

$$pPVARP(X_s;\ p\_vp, s\_vp, vsp\_pace;\ p\_pvarp\_ended;\ vs, vsp\_pace;\ TP_1(DLY, C(pPVARP\_t_{DLY})), TP_2(DDL, C(pPVARP\_t_{DDL}))) \tag{9.15}$$

$$sPVARP(X_s;\ INIT, vs;\ s\_pvarp\_ended;\ vs, vsp\_pace;\ TP_1(DLY, C(sPVARP\_t_{DLY})), TP_2(DDL, C(sPVARP\_t_{DDL}))) \tag{9.16}$$

pacemaker may still sense signals that could influence certain timing cycles such as automatic mode switching (not covered in this work); blanking periods overcome this by preventing sensing whatsoever [35].

In the final refinement, we introduce four blanking periods to paced refractory intervals: *ventricular blanking period* (VBP) for $pVRP$; *post ventricular atrial blanking period* ($PVABP$) for $pPVARP$; parallel *atrial blanking period* (ABP) and *post-atrial ventricular blanking period* ($PAVBP$) for VSP_atr. Each abstract interval is refined into a sequence of sub-intervals, where the first part is taken up by the blanking period.

We firstly refine interval *VSP_atr* to two parallel sub-interval sequences (Figure 9.22). The first sequence consists of sub-intervals *PAVBP* (9.17) and *PAVBP_off* (9.18), the second – *ABP* (9.19) and *ABP_off* (9.20). The refinement process for both sequences is analogous to that described in subsection 9.2.3.

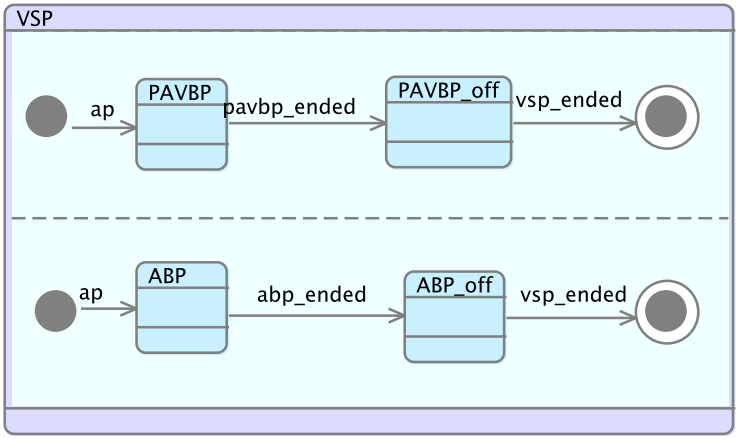

Figure 9.22: 6th ref.: interval *VSP* refined to PAVBP and ABP sub-interval sequences.

$$PAVBP(X_s;\ ap;\ pavbp\_ended;\ --;\\ TP_1(DLY, C(PAVBP\_t_{DLY})), TP_2(DDL, C(PAVBP\_t_{DDL}))) \tag{9.17}$$

$$PAVBP\_off(X_s;\ pavbp\_ended;\ vsp\_ended, vsp\_pace;\\ TP_1(DLY, C(PAVBP\_off\_t_{DLY})), TP_2(DDL, C(PAVBP\_off\_t_{DDL}))) \tag{9.18}$$

$$ABP(X_s;\ ap;\ abp\_ended;\ --;\\ TP_1(DLY, C(ABP\_t_{DLY})), TP_2(DDL, C(ABP\_t_{DDL}))) \tag{9.19}$$

$$ABP\_off(X_s;\ abp\_ended;\ vsp\_ended, vsp\_pace;\\ TP_1(DLY, C(ABP\_off\_t_{DLY})), TP_2(DDL, C(ABP\_off\_t_{DDL}))) \tag{9.20}$$

Both sequences are triggered and responded to by abstract interval *VSP_atr* trigger and response events (TIME - 38, TIME - 42). Neither *PAVBP* nor *ABP* can be aborted (TIME - 43). Consistency between both sub-interval sequence durations and the abstract interval *VSP_atr* duration is ensured with axioms (TIME - 40, TIME - 41, TIME - 44, TIME - 45).

Similarly, we elaborate *pVRP* with the sub-interval sequence (Figure 9.23) of *VBP* (9.21) and *VBP_off* (9.22); and abstract interval *pPVARP* with sub-interval sequence (Figure 9.24) of *PVABP* (9.23) and *PVABP_off* (9.24).

Both sequences share trigger and response events with their abstract intervals (TIME - 34, TIME - 46). Blanking intervals *VBP* and *PVABP* cannot be aborted (TIME - 35, TIME - 47). Duration consistencies are ensured with axioms (TIME - 36, TIME - 48, TIME - 37, TIME - 49).

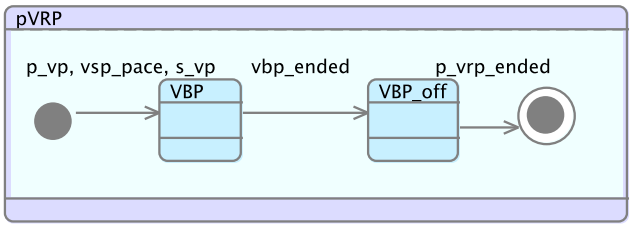

Figure 9.23: VBP

$$VBP(X_s;\ p\_vp, s\_vp, vsp\_pace;\ vbp\_ended;\ --;\\ TP_1(DLY, C(VBP\_t_{DLY})), TP_2(DDL, C(VBP\_t_{DDL}))) \tag{9.21}$$

$$VBP\_off(X_s;\ vbp\_ended;\ p\_vrp\_ended;\ as;\\ TP_1(DLY, C(VBP\_off\_t_{DLY})), TP_2(DDL, C(VBP\_off\_t_{DDL}))) \tag{9.22}$$

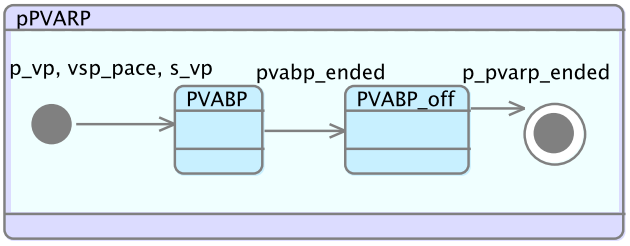

Figure 9.24: PVABP

$$PVABP(X_s;\ p\_vp, s\_vp, vsp\_pace;\ pavbp\_ended;\ --;\\ TP_1(DLY, C(PVABP\_t_{DLY})), TP_2(DDL, C(PVABP\_t_{DDL}))) \tag{9.23}$$

$$PVABP\_off(X_s;\ pavbp\_ended;\ p\_pvabp\_ended;\ vs, vsp\_pace;\\ TP_1(DLY, C(PVABP\_off\_t_{DLY})), TP_2(DDL, C(PVABP\_off\_t_{DDL}))) \tag{9.24}$$

9.3 Overview and Conclusions

We have successfully modelled the pacemaker following the two-phase approach as described in chapter 7. Firstly, we have modelled the functional aspects of the system and then added the timing dimension with the new timing interval approach on top.

The functional part of the system has been expressed visually in state machines via the iUML tool. The timing requirements were generated on top with the tiGen tool. All timing intervals were generated automatically except for the guards that pre-set dynamic durations in the trigger events. Consistency invariants to relate timing intervals to corresponding state machine states were added manually as the iUML and tiGen integration has not yet been done.

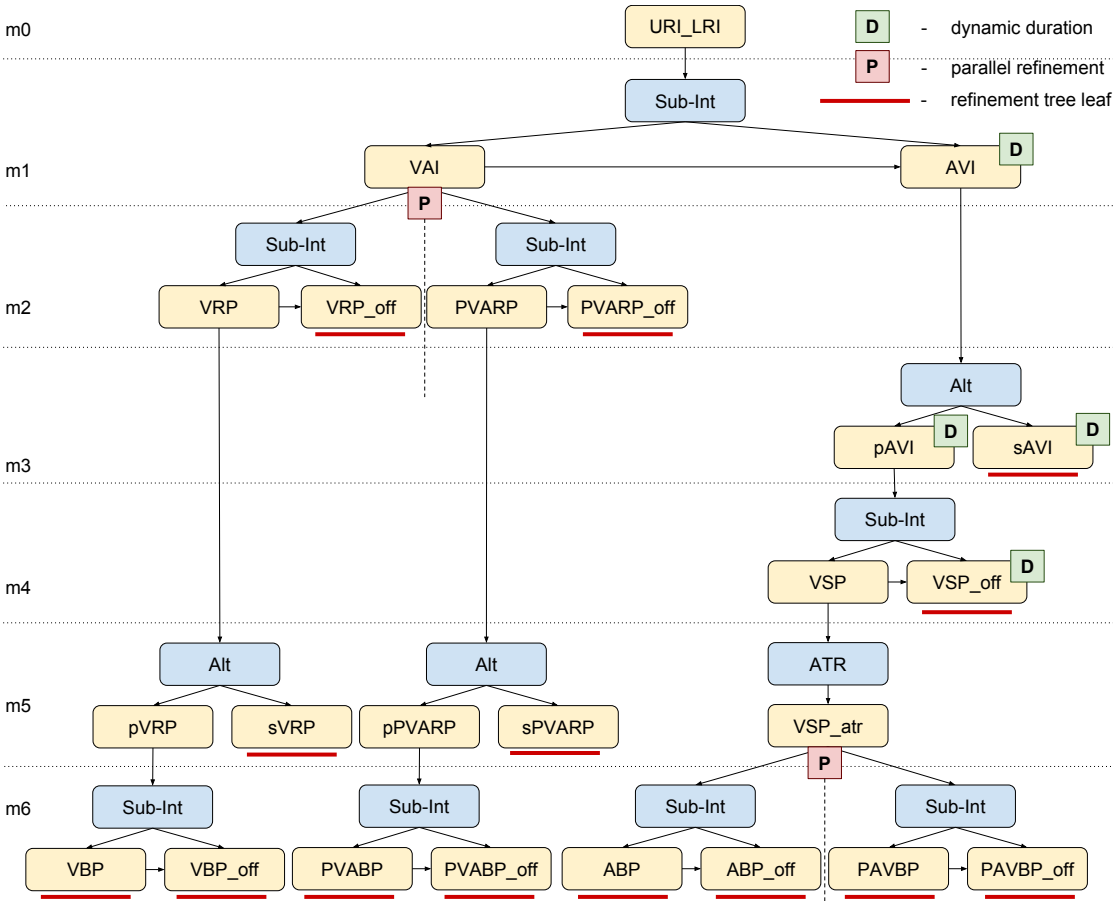

Figure 9.25: Refinement tree of six refinement levels.

We have demonstrated the abort functionality, multiple trigger and response support, overloaded events. We have validated our approach through six refinement levels (Figure 9.25), made use of Alternative, Sub-Interval, Abort-to-Response patterns and the parallel refinement feature. Figure 9.25 provides the outline of the refinement structure. We have started with interval *URI_LRI* at the abstract machine and elaborated it to 14 concrete intervals, underlined in red. We have used Sub-Interval (Sub-Int), Alternative (Alt) and Abort-to-Response (ATR) refinement transformations.

We have used a fine-tuned set of prover tactics (section 8.1) to discharge generated proof obligations (POs). All 1571 proof obligations were discharged automatically (Table 9.1). The final model has 85 variables, 76 invariants and 18 events. A total of 248 invariants were generated across all refinement levels.

We have written eight test case scenarios (Appendix C) to validate intrinsic and artificial atrial and ventricular channel activity. Scenarios were executed in ProB animator with positive results. Each test scenario tackles a specific case, e.g. or "**a**. Testing the occurrence of atrial sense after any ventricular event has occurred" or "**b**. Testing

Ref. Lvl	POs	Manual POs	Variables	Invariants	Events
m0	53	0	8	15	5
m1	159	0	17	30	7
m2	205	0	29	36	9
m3	154	0	39	27	10
m4	134	0	47	24	12
m5	332	0	61	40	14
m6	533	0	85	76	18
Total	**1571**	**0**	85	248	18

Table 9.1: Proving statistics.

ventricular sense after atrial sense has occurred". We can then form words of these test cases to test a more sophisticated scenario. For example word a,b,a,b,a,b would test the alternation of ventricular and atrial sense event occurrences, starting with the ventricular sense (a). However, manual test case scenarios can cover a fraction of all possible cases and validate only a subset of model states.

Finally, we have used ProB model checker to perform a full-coverage model-checking using close to realistic interval values (listen in subsection D.1.13). No deadlocks or inconsistencies were found.

Chapter 10

Message Passing Case Study

In this chapter, we investigate the applicability of the timing interval approach by modelling a message passing protocol. We describe the system and narrate the requirements along with the details on the formal model. We show how the timing interval approach copes with multi-instance models and what limitations there are to address in the future work. Finally, we conclude with the results.

10.1 Message Transfer Protocol

A message passing protocol defines a message transfer process between two parties over a communications channel. The protocol consists of three actors: a sender, a communications channel and a receiver. We overview each actor in greater detail and provide assumptions about the environment that the protocol is expected to work in.

Sender. The sender is the only proactive actor in our considered system. It can initiate multiple independent message transfers at any point in time and is responsible for handling faults, such as lost packets, that may occur in other actors. The sender itself is assumed to be ideal hence no errors can occur on its side.

Communication channel. We define the communications channel as an abstract medium between the sender and the receiver. It may be a physical communications link, e.g. radio waves or a wire, managed by a piece of unspecified software and hardware. We do not model network in this case study but take into consideration that this is a full-duplex capable channel. In the full duplex system, both the sending and the receiving parties can communicate with each other simultaneously.

Receiver. The receiver is stateless and reactive. It does not track sent or received message packets. Upon receiving a packet, it simply sends the response back to the sender.

The message transfer process, displayed in Figure 10.1, can be broken down into three stages: 1. the pre-processing of the message to be transferred; 2. the actual message transfer; 3. post-processing the response data; (4.) the message transfer comprises a number of packet transfer processes that can be sent out concurrently to the receiver over the communications channel. Each packet process can be further broken down into packet-transfer attempts. If the packet-transfer process does not successfully complete within a predefined duration of time, the sender handles the fault by retrying the packet transfer process again. A packet-transfer attempt consists of: 5. the packing of a packet to be sent, 6. the actual packet transfer, 7. processing the received response packet. The message is considered to be sent when the sender successfully completes all packet transfers, and the received data is processed (3.)

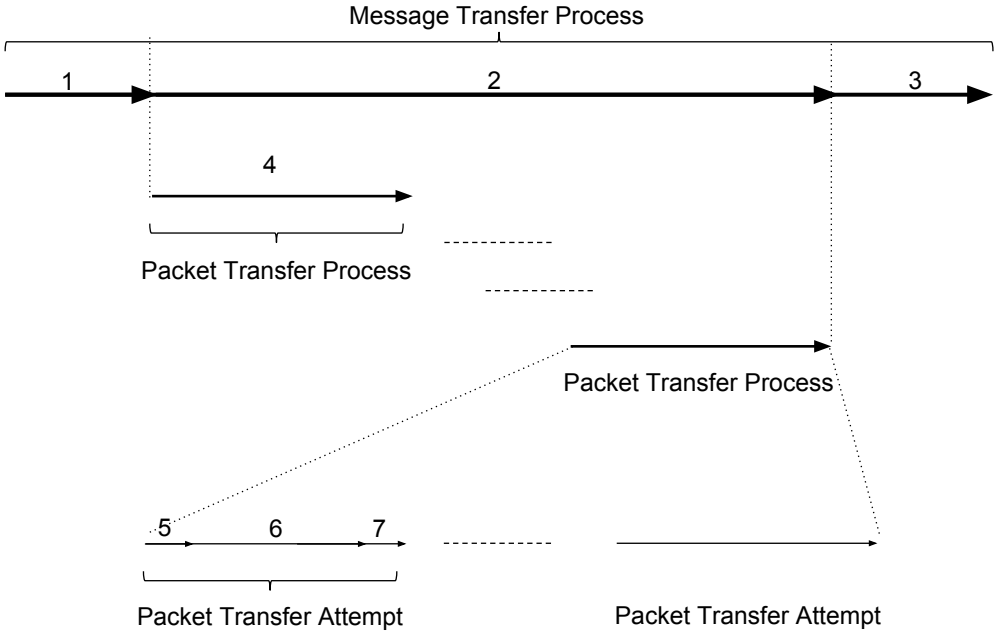

Figure 10.1: Message transfer process break-down.

During the transmission, the packets may be lost. Since we model only the sender side, we are not concerned where and how the data is lost, e.g. due to a faulty communications channel or an error on the receiver's end. We are concerned only with the fact that the packet times out before it reaches the sender. The sender can handle possible data loss to some degree. After a certain number of lost packets, the system enters an unrecoverable error state.

A list of requirements for the protocol is categorised into three groups (section D.2): environment assumptions (ENV), functional (FUN), timing (TIME) and error (ERR)

requirements. A compiled list of requirements can be found in section D.2. We refer to the requirements along the narration of the modelling process.

10.2 Modelling Strategy and Highlights

The system has been modelled according to the guidelines outlined in chapter 7: we firstly generate iUML state machines and then superimpose timing intervals using the tiGen tool. The timing intervals and their refinements are visually represented as per convention, as given in the guidelines chapter. The model has resulted in six refinements and consists of several state machines and timing intervals, all of which are multi-instance. We highlight manually added Event-B elements. The semantics of the multi-instance state machine is explained in chapter 3.

In this case study, we demonstrate the multi-instance timing interval and its synchronisation with a multi-instance state machine. We make use of Retry and Single-to-Multi refinement transformations and demonstrate their interoperability with Sub-Interval and Abort-to-Response transformations. We used only delay and deadline timing properties in the case study.

Timing property constraints are applied on all timing interval response events. In this case study, we show how only specified response events may be further constrained by using additional timing intervals. This is performed manually and is part of the future work to be automated.

10.2.1 Abstract Machine: Message Transfer

We start by modelling the multi-process sender, which can send multiple messages concurrently (Figure 10.3). Semantically, the state machine is represented as function *SM_msg* (Figure 10.2), which records each message's active state. Set *MESSAGES* represents available messages, and set *msg_STATES* defines possible message states.

$$\texttt{inv1}: \ \texttt{SM\_msg} \in \texttt{MESSAGES} \rightarrow \texttt{msg\_STATES}$$

Figure 10.2: AUTO generated by iUML: state machine *SM_msg* type.

All message transfer processes start in the *msg_ready* state. The state denotes the sender is ready to start the message transfer. The following events can act independently for every message available in set *MESSAGES* (ENV-1). Event *msg_start* initiates the message transfer process (FUN-1). State *msg_transferring* denotes the message transfer process in progress. In case of an unrecoverable error, the system enters error

state *msg_error* (ERR-1). Otherwise the process successfully completes and the message transfer reaches *msg_finished* state (ENV-2, ENV-3). Upon transitioning to state *msg_finished*, we say that the sender has processed the last response packet and the transfer has been successful (FUN-4). The message can be prepared to be re-sent in case of an unrecoverable error or a finished transfer with event *msg_reset*.

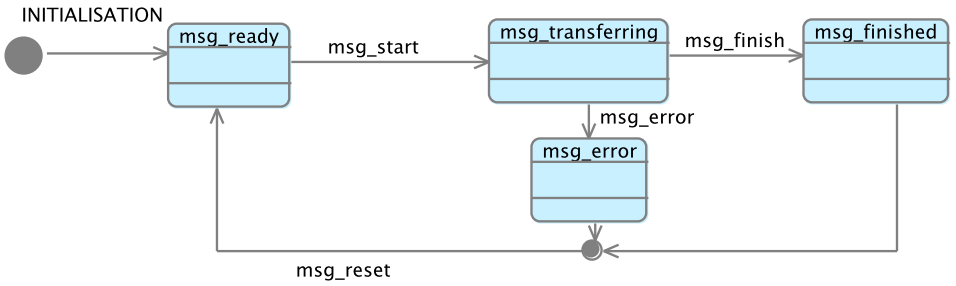

Figure 10.3: Multi-instance state machine *SM_msg*.

$$MSG(MESSAGES, msg\_start;\ msg\_finish;\ msg\_error;\ TP_1(DDL, C(MSG\_t_{DDL}))) \tag{10.1}$$

The timing interval *MSG* (10.1) places a deadline constraint for every message in the *MESSAGES* set, requiring the message transfer to finish within time MSG_t_{DDL} TIME-4. The timing interval is triggered by the *msg_start* event and is responded to by the *msg_finish* event. An error can occur at any point in time, consequently aborting the interval.

We synchronise state machine and timing interval instances with invariant (Figure 10.4). The invariant says that for every triggered *MESSAGE* instance that has not yet been aborted or responded to there is a state machine in active message transfer state. Where *msgSelf* is the state machine instance in question, and *p_MSG_trig* is the trigger parameter for the *MESSAGE* timing interval. Similarly, we synchronise state machine and timing interval indices in response and abort events *msg_finish* and *msg_error* respectively.

$$inv1 : \forall idx \cdot idx \in MSG\_trig \setminus (MSG\_resp \cup MSG\_abrt) \Leftrightarrow msg(idx) = msg\_transferring$$

Figure 10.4: MANUALLY added: timing interval *MSG* and state machine *SM_msg* synchronisation invariant.

A *Tick* event, not displayed in the diagram, is used to model the time-flow. All timing intervals introduced into the model rely on this event to model the time-flow. We use event *msg_reset* to reset timing interval indices for timing interval *MSG* and others, as discussed in this case study.

10.2.2 First Refinement: Message Transfer Sub-tasks

In the first refinement, we introduce the three message transfer stages (FUN-2). We do so by adding state machine *SM_msg_transferring* (Figure 10.5). We follow the convention to name sub-state machines by their parent state names by prefixing them with *SM_*. The message pre-process stage is represented by state *msg_pre_process*; the actual message transfer stage represented by state *msg_transfer*; *msg_post_process* denotes the post processing of the successfully received message. Since we assume that the sender does not have faults (ENV-5), the error can occurr only during the transfer process while in state *msg_transfer*.

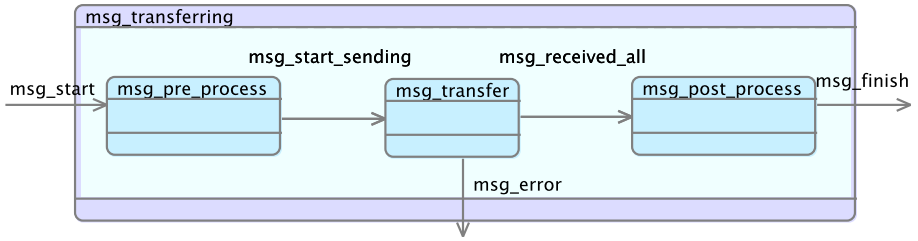

Figure 10.5: State-machine *SM_msg_transferring*: message transfer stages.

$$MSG\_pre(MSG, msg\_start;\ msg\_start\_sending;\ --;\ TP_1(DDL, C(MSG\_pre\_t_{DDL}))) \tag{10.2}$$

$$MSG\_tr(MSG, msg\_start\_sending;\ msg\_received\_all;\ msg\_error;\ TP_1(DDL, C(MSG\_tr\_t_{DDL}))) \tag{10.3}$$

$$MSG\_post(MSG, msg\_received\_all;\ msg\_finish;\ --;\ TP_1(DDL, C(MSG\_post\_t_{DDL}))) \tag{10.4}$$

We refine the *MESSAGE* timing interval into three sub-intervals by applying Sub-Interval refinement transformation (subsection 5.1.2). Sub-interval *MSG_pre* (10.2) denotes the message pre-process time period. The interval is triggered by event *msg_start*. The process must complete within time $MSG\_pre\_t_{DDL}$ and at the end of the interval respond by triggering the message transfer process (TIME-1).

The message transfer process *MSG_tr* (10.3), triggered with event *msg_start_sending*, must receive all packet responses within time $MSG\_tr\_t_{DDL}$ (TIME-2). While active, the timing interval may be aborted due to possible faults. After the message has been transferred, the sender enters the post-processing stage.

After the transfer has been successfully completed, the message post-process interval *MSG_post* is triggered by event *msg_received_all*. The timing interval must complete within time $MSG\_post\_t_{DDL}$ (TIME-3).

Note that pre and post processing cannot be aborted by *msg_error*. This is due to the assumption that no faults may occur in the sender (ENV-5). Both the functional

and the timing parts have been automatically generated with iUML and tiGen tools respectively.

10.2.3 Second Refinement: Packet Transfer

In the second refinement, we elaborate the message transfer with the packet transfer process. A new root multi-instance state machine is added to represent the packet transfer process in detail (Figure 10.6).

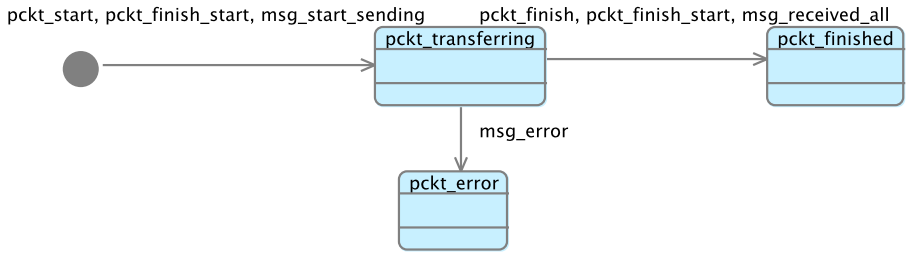

Figure 10.6: Multi-instance packet transfer state machine *SM_pckt*.

$$
\begin{aligned}
&PACKET(MSG \times PCKT;\\
&\qquad \textbf{pckt\_finish\_start}, \textbf{pckt\_start}, msg\_start\_sending;\\
&\qquad \textbf{pckt\_finish\_start}, \textbf{pckt\_finish}, msg\_received\_all;\\
&\qquad msg\_error;\\
&\qquad TP_1(DDL, C(PCKTS\_t_{DDL})))
\end{aligned}
\tag{10.5}
$$

Semantically, each packet has its state recorded in function *SM_pckt* (Figure 10.7). Where the Cartesian product is a set of all available packets for every message and set *pckt_STATES* holds possible packet states. For simplicity, all messages have the same number of packets.

```
inv1 : SM_pckt ∈ (MESSAGES × PACKETS) → pckt_STATES
```

Figure 10.7: AUTO generated by iUML: state machine *SM_pckt* representing a packet transfer.

Every packet can be in one of four states (Figure 10.6): active transfer state *pckt_transferring*, successfully transferred – *pckt_finished*, irrecoverable – *pckt_error*, or not yet started – *NULL*. The latter is not represented in the figure.

The first packet transfer is always triggered by the *msg_start_sending* event. Event *pckt_finish_start* represents the case when a response packet is received, and the sender immediately sends out a new packet if such is available (TIME-11). If there are no packets to send, the sender simply receives the response with event *pckt_finish*. Multiple

packets can be in transfer at any point in time (FUN-6). We use event *pckt_start* to send new packets. Packets can be sent and received in any order (FUN-13, FUN-14) regardless of when they have been sent. Event *msg_received_all* receives the response packet of the last packet transfer process.

We elaborate state *msg_transferring* of state machine *SM_msg* (Figure 10.3) with a sub-state machine *SM_msg_transfer*. The new sub-state machine contains one state *msg_transferring_pckts*. The state denotes the fact that there is at least one active packet transfer process. State-machine *SM_msg_transfer* can be in one of two states. In case its parent state *msg_transfer* is not active, the state machine is in *NULL* state, otherwise – in *msg_transferring_pckts*.

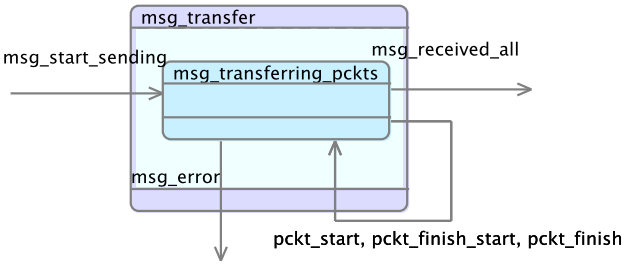

Figure 10.8: State-machine *SM_msg_transfer*: elaborated message passing process.

Notice the shared events between the two state machines. To ensure that the message and packet transfer state machines operate in synchrony, we have added three equivalence invariants (Figure 10.9). Invariant *inv*1 ensures that when the message is in active transfer, at least one packet process must be active; if the message transfer process is post-processing or complete, that means that all message-related packets have been successfully transferred (*inv*2); finally, if message transfer is in the unrecoverable state, then at least one packet is in the error state as well (*inv*3).

$$
\begin{aligned}
&\mathit{inv1}: \forall m \cdot m \in MSG \land SM\_msg\_transfer(m) = msg\_transferring\_pckts \Leftrightarrow \\
&\quad (\exists p \cdot p \in PCKT \land pckt(m \mapsto p) = pckt\_transferring) \\
&\mathit{inv2}: \forall m \cdot m \in MSG \land ((SM\_msg\_transferring(m) = msg\_post\_process) \lor (msg(m) = msg\_finished)) \Leftrightarrow \\
&\quad pckt[\{m\} \times PCKT] = \{pckt\_finished\} \\
&\mathit{inv3}: \forall m \cdot msg(m) = msg\_error \Leftrightarrow \\
&\quad (\exists p \cdot p \in PCKT \Rightarrow pckt(m \mapsto p) = pckt\_error)
\end{aligned}
$$

Figure 10.9: MANUALLY added: *SM_msg* and *SM_pckt* state machine synchronisation invariants.

In case of the error, event *msg_error* changes active packets' state to *pckt_error* and the whole message transfer status to *msg_error*. The new message transfer process can no longer send new packets.

We refine the *MSG_tr* timing interval to *PCKT* (10.5) by applying the Single-to-Multi refinement transformation (subsection 5.1.4). Each sub-instance of the new interval represents a packet transfer process. Each packet transfer must complete within time $PCKT\_t_{DDL}$ (TIME-8), where $MSG\_tr\_t_{DDL} = card(PACKETS) * PCKT\_t_{DDL}$ | and $card(PACKETS)$ is the number of packets processes in the message.

The first packet is sent with event *msg_start_sending*. Subsequent packets are sent with *pckt_start* or the trigger-response event *pckt_finish_start*. The intermediate packets are responded to by the *pckt_finish* event and the last packet is responded to by *msg_received_all*. In the event of an error, all message-related packet transfer sub-instances are to be aborted. Newly introduced events are marked in bold.

The timing interval is synchronised with the new state machine with an invariant, displayed in (Figure 10.10).

$$\texttt{inv1}: \forall \texttt{idx} \cdot \texttt{idx} \in \texttt{PCKTS\_trig} \setminus (\texttt{PCKTS\_resp} \cup \texttt{PCKTS\_abrt}) \Leftrightarrow \texttt{pckt(idx)} = \texttt{pckt\_transferring}$$

Figure 10.10: MANUALLY added: timing interval *PCKTS* and state machine *SM_pckt* synchronisation invariant.

10.2.4 Third Refinement: Packet Retries

The packet transfer process is transactional – a packet is considered as transferred once the sender receives the message response for the sent packet. In case the sender does not receive the response, the packet transfer process is re-sent. This dynamic is represented in the *SM_pckt_transferring* sub-state machine (Figure 10.11).

The packet transfer process starts in state *pckt_transfer*. Every packet starts with a zero resend count, recorded in function *retryCount* (Figure 10.12). Where *MaxResend* is the maximum allowed number of packet retries (FUN-8). The transfer process can either complete or time-out (ENV-4). In instances of the latter, the packet transfer process enters the *pckt_retrying* state and its *retryCount* is incremented by one. If the retry count does not exceed the limit, event *pckt_retry* triggers a packet re-transfer process.

In the case when the *MaxResend* limit is reached, the packet transfer is put into *pckt_r_limit_reached* state (ERR-1). When there is at least one packet in this state, new packet transfers are not allowed, but the existing packet transfers may complete by the time the sender enters the unrecoverable message transfer error state (FUN-9).

Five manually added invariants define the relation between the *retryCount* variable packet transfer states. Invariant *inv*1 ensures that when the message transfer enters the error state, at least one of the packets is erroneous. Invariant *inv*2 states that all

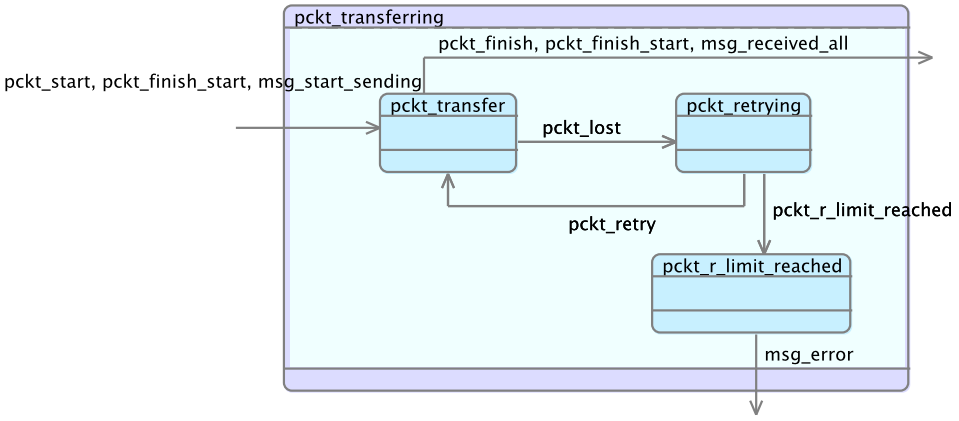

Figure 10.11: State-machine *SM_pckt_transferring*: elaborated packet transfer process.

$$
\begin{aligned}
&RETRY(MSG \times PCKT \times RETRIES;\\
&\quad msg\_start\_sending, pckt\_finish\_start, \mathbf{pckt\_retry}, pckt\_start;\\
&\quad msg\_received\_all, pckt\_finish, pckt\_finish\_start;\\
&\quad msg\_error, \mathbf{pckt\_retry};\\
&\quad TP_1(DDL, C(RETRY\_t_{DDL})))
\end{aligned}
\tag{10.6}
$$

$$\text{inv1}: \ \text{retryCount} \in (\text{MESSAGES} \times \text{PACKETS}) \rightarrow 0\,..\,\text{MaxResend}$$

Figure 10.12: MANUALLY added: retry count function for every packet of each message.

active packet transfers, that are not in the error state, must not exceed the resend count. Invariant *inv*3 ensures that all non-started packet transfers have zero transfer counts. The last two invariants synchronise *MSG_tr* and *PCKT* state machine states. Invariant *inv*4 ensures that when the message transfer enters the error state, none of the relevant packet transfer can be active. The last invariant *inv*5 ensures that when the message transfer is finished, all relevant packet transfer processes are set to *NULL*.

$$\begin{aligned}
&\text{inv1}: \ \forall m \cdot msg(m) = msg_error \Rightarrow (\exists p \cdot retryCount(m \mapsto p) = RETRY_COUNT)\\
&\text{inv2}: \ \forall p \cdot p \in dom(pckt) \wedge pckt(p) \neq pckt_error \Rightarrow retryCount(p) \leq RETRY_COUNT\\
&\text{inv3}: \ \forall p \cdot pckt(p) = pckt_NULL \Rightarrow retryCount(p) = 0\\
&\text{inv4}: \ \forall m \cdot msg(m) = msg_error \Rightarrow SM_pckt_transferring[\{m\} \times PCKT] \not\subseteq \{pckt_transfer\}\\
&\text{inv5}: \ \forall m \cdot msg(m) = msg_finished \Rightarrow SM_pckt_transferring[\{m\} \times PCKT] = \{SM_pckt_transferring_NULL\}
\end{aligned}$$

Figure 10.13: MANUALLY added: *PCKT* state machine synchronisation with retries.

We refine the *PCKT* timing interval to interval *RETRY* (10.6) with Retry refinement transformation (5.1.5). A packet response is expected to arrive within time $RETRY\_t_{DDL}$. If not, the packet is considered lost and the transfer process is repeated again for up to *MaxResend* times (TIME-9). The duration of packet transfer

attempts must be consistent with the packet transfer process duration – $PCKT\_t_{DDL} = MaxResend * RETRY\_t_{DDL}$.

10.2.5 Fourth Refinement: Packet Time-out

In the fourth refinement, we transform *msg_error* and *pckt_retry* events to responses, thus forcing the packet transfer process re-try and error occurrence within the $RETRY\_t_{DDL}$ time (10.7).

$$\begin{aligned} &RETRY\_atr(MSG \times PCKT \times RETRIES; \\ &\quad msg\_start\_sending, pckt\_finish\_start, pckt\_retry, pckt\_start; \\ &\quad \textbf{msg\_error}, msg\_received\_all, pckt\_finish, pckt\_finish\_start, \textbf{pckt\_retry}; \\ &\quad --; \\ &\quad TP_1(DDL, C(RETRY\_t_{DDL}))) \end{aligned} \tag{10.7}$$

We add a delay restriction on the packet retry by adding a Time-Out (TO) timing interval. A packet can be resent if at least time TO_t_{DLY} time-units has passed from the previous attempt (TIME-10). Where $TO\_t_{DLY} = RETRY\_t_{DLY}$

$$\begin{aligned} &TO(MSG \times PCKT \times RETRIES; \\ &\quad msg\_start\_sending, pckt\_finish\_start, pckt\_retry, pckt\_start; \\ &\quad pckt\_retry; \\ &\quad msg\_error, msg\_received\_all, pckt\_finish, pckt\_finish\_start; \\ &\quad TP_1(DLY, C(TO\_t_{DLY}))) \end{aligned} \tag{10.8}$$

TO and $RETRY\_atr$ synchrony is ensured via invariant (10.14).

```
inv1 : RETRY_trig \ (RETRY_resp ∪ RETRY_abrt) = TO_trig \ (TO_resp ∪ TO_abrt)
```

Figure 10.14: MANUALLY added: timing interval $PCKT$ and TO synchronisation invariant.

10.2.6 Fifth Refinement: Packet Resend Constraint

In the fifth refinement, we elaborate interval *TO* with two sub-intervals. The first part represents the delay requirement before the packet may time-out. Note that we reuse the same delay duration as the abstract interval.

$$\begin{aligned} &TO\_delay(MSG \times PCKT \times RETRIES; \\ &\quad msg\_start\_sending, pckt\_finish\_start, pckt\_retry, pckt\_start; \\ &\quad pckt\_lost; \\ &\quad msg\_error, msg\_received\_all, pckt\_finish, pckt\_finish\_start; \\ &\quad TP_1(DLY, C(TO\_t_{DLY}))) \end{aligned} \tag{10.9}$$

The second sub-interval defines the requirement that in case of the time-out, a packet must be resent immediately (TIME-11). Hence the delay and deadline duration equal to zero.

$$\begin{aligned} &TO\_resend(MSG \times PCKT \times RETRIES; \\ &\quad msg\_start\_sending, pckt\_finish\_start, pckt\_retry, pckt\_start; \\ &\quad pckt\_retry; \\ &\quad msg\_error, msg\_received\_all, pckt\_finish, pckt\_finish\_start; \\ &\quad TP_1(DLY, C(0)), TP_2(DDL, C(0))) \end{aligned} \tag{10.10}$$

10.2.7 Sixth Refinement: Packet Pre and Post Processing

In the last refinement, we break down a packet transfer into three stages: packet pre-process (FUN-11), the actual packet transfer, and packet response post-processing. We reflect the three stages in an *SM_pckt_transfer* sub-state machine. Where state *pckt_packing* represents preparing a packet to be sent out, actual packet transfer *pckt_tr*, and received packet response unpacking *pckt_unpacking*. State *pckt_not_received* indicates the fact that the packet has been lost in transmission.

On top of the generated state machine, we superimpose three sub-intervals that together elaborate the abstract timing interval *RETRIES_atr*. The first sub-interval *PCKT_pre* (10.11) expresses the requirement that the packet pre-processing cannot take more than time $PCKT\_pre\_t_{DDL}$. The sub-interval is triggered by the same trigger events as the abstract interval *RETRY*. When the packet is prepared for transfer, it is then sent to the communications channel buffer, represented by event *pckt_send_packet*.

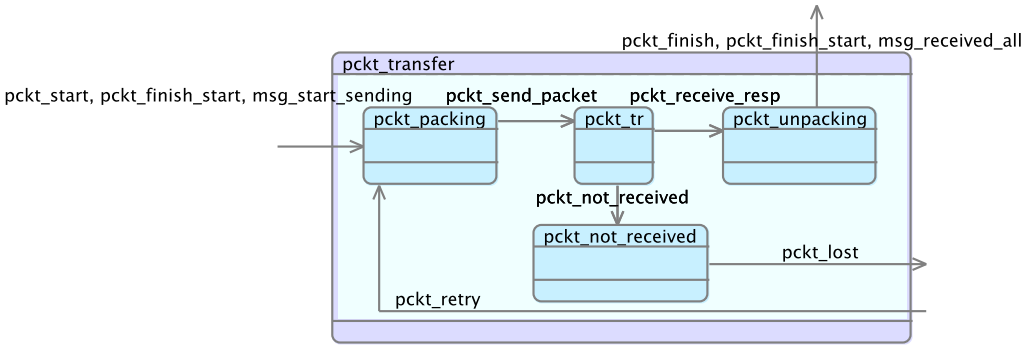

Figure 10.15: State-machine *SM_pckt_transfer*: packet transfer stages.

$$
\begin{aligned}
&PCKT\_pre(MSG \times PCKT \times RETRIES;\\
&\quad msg\_start\_sending, pckt\_finish\_start, pckt\_retry, pckt\_start;\\
&\quad pckt\_send\_packet; \qquad (10.11)\\
&\quad --;\\
&\quad TP_1(DDL, C(PCKT\_pre\_t_{DDL})))\\
&PCKT\_tr(MSG \times PCKT \times RETRIES;\\
&\quad pckt\_send\_packet;\\
&\quad pckt\_receive\_resp; \qquad (10.12)\\
&\quad --;\\
&\quad TP_1(DDL, C(PCKT\_tr\_t_{DDL})))\\
&PCKT\_post(MSG \times PCKT \times RETRIES;\\
&\quad pckt\_receive\_resp;\\
&\quad msg\_error, msg\_received\_all, pckt\_finish, pckt\_finish\_start, pckt\_retry;\\
&\quad --;\\
&\quad TP_1(DDL, C(PCKT\_post\_t_{DDL})))\\
&\qquad\qquad (10.13)
\end{aligned}
$$

Interval *PCKT_tr* (10.12) expresses the requirement that the packet transfer round-trip from the sender to the receiver and back takes no more than $PCKT\_tr\_t_{DDL}$ time. When the packet arrives, the transfer process enters the third stage.

In the third stage, the sender has to process the response packet within time $PCKT\_post\_t_{DDL}$. This is ensured by timing interval *PCKT_post* (10.13).

Similarly, we elaborate abstract interval *TO_delay* to three sub-intervals. Interval *TO_pre* (10.14) defines the pre-process stage which, in our case, is instantaneous – delay and deadline durations are equal to zero.

$$
\begin{aligned}
&TO\_pre(MSG \times PCKT \times RETRIES;\\
&\quad msg\_start\_sending, pckt\_start, pckt\_finish\_start, pckt\_retry;\\
&\quad pckt\_send\_packet;\\
&\quad msg\_error, msg\_received\_all, pckt\_finish, pckt\_finish\_start;\\
&\quad TP_1(DDL, C(0)), TP_2(DDL, C(0)))
\end{aligned}
\tag{10.14}
$$

Interval *TO_tr* (10.14) places a delay constraint on event *pckt_not_received* for the duration of the packet transfer while in state *pckt_tr*. The sender is required to wait for the $TO\_tr\_t_{DLY}$ time until the transfer may time-out. Otherwise, if the packet response is received, the interval is aborted later after the packet has been post-processed.

$$
\begin{aligned}
&TO\_tr(MSG \times PCKT \times RETRIES;\\
&\quad pckt\_send\_packet;\\
&\quad pckt\_not\_received;\\
&\quad msg\_error, msg\_received\_all, pckt\_finish, pckt\_finish\_start;\\
&\quad TP_1(DLY, C(TO\_tr\_t_{DLY})))
\end{aligned}
\tag{10.15}
$$

If the packet has timed-out, timing interval *TO_post* (10.16) is triggered, ensuring that the packet transfer process is immediately declared as lost with the response event *pckt_lost*.

$$
\begin{aligned}
&TO\_post(MSG \times PCKT \times RETRIES;\\
&\quad pckt\_not\_received;\\
&\quad pckt\_lost;\\
&\quad msg\_error, msg\_received\_all, pckt\_finish, pckt\_finish\_start;\\
&\quad TP_1(DDL, C(0)), TP_2(DDL, C(0)))
\end{aligned}
\tag{10.16}
$$

10.3 Verification and Validation

A total of 1015 proof obligations were generated (Table 10.1). All time-related POs were discharged automatically using our fine-tuned auto prover (section 8.1). Two POs related to the functional part of the system had to discharged manually. The final model has 64 variables, 32 invariants and 27 events. A total of 139 invariants were generated across all refinement levels.

Ref. Lvl	POs	Manual POs	Variables	Invariants	Events
m0	51	0	7	11	6
m1	101	0	16	19	8
m2	187	0	25	23	11
m3	274	0	35	30	14
m4	107	0	43	15	14
m5	58	0	47	9	14
m6	231	0	64	32	27
Total	**1015**	**0**	64	139	27

Table 10.1: Proving statistics.

The model with a single message ($card(MSG) = 1$) has been fully model checked and no deadlocks or inconsistencies were found. Due to a large state-space, the model with undefined set values has been only partially model-checked. Finally, we animated the model with the ProB animator and found no inconsistencies.

10.4 Comparison to Sarshogh's Message Passing Case Study

The message passing protocol case study, described in this chapter, is based on Sarshogh's message transfer system. We point out key differences between the two in terms of design and suitability to modelling time requirements.

In our case study, the message transfer process is transformed into parallel packet transfers. The packets are sent and received simultaneously in an arbitrary order . The modelling process is split into two iterations according to the guidelines in section 7.1. In the first iteration we describe only the functional part of the system and in the second iteration, we superimpose explicit time using the timing interval approach. Sarshogh refines the abstract message transfer process into a strict sequence of packet transfers. The order of the packets being transferred is encoded with the timing approach. The tight coupling of the packet ordering, which is a functional aspect, and the time hinders the two-iteration modelling process. The lack of separation of concerns in Sarshogh's approach makes it more difficult to model, verify and validate the system.

The higher-level notion of timing interval makes it intuitive to reason about the message transfer process as an entity with delay and deadline timing properties. The message transfer process can then be gradually refined into packet transfers using the provided refinement transformations (chapter 5). In Sarshogh's case, the packet transfer system is a collection of separately refined deadlines and delays, each having different refinement structure.

The timing interval produces a finite-state space model, allowing to potentially fully model-check the model with the timing part. In the case of Sarshogh's approach, once the time is introduced, the model state-space becomes infinite due to the unbounded *time* variable that models the time.

We did not investigate the decomposition of the timing interval and thus cannot compare this aspect to Sarshogh's approach. We acknowledge that this feature introduces a major advantage in tackling the complexity of large models.

10.5 Overview and Conclusions

We have modelled the message passing case study with the two-phase approach, and the final model resulted in six refinements. The system functional requirements have been expressed in iUML state machine diagrams. We have demonstrated the Single-to-Multi and Retry refinement transformations and that they are compatible with Sub-Interval and Abort-to-Response refinement transformations. Customizations to the generated code were needed to synchronise the message and packet state machine instances to the corresponding timing intervals. The timing dimension has been superimposed on the abstract model using our timing interval generation tool. Custom code was needed to relate timing interval instances to packet transfer state machine instances.

The timing interval refinement tree is displayed in Figure 10.16. Refinement leaves are marked with red underlines. Single-to-Multi refinement transformation is marked as *STM*, and Retry – as *RTR*.

The model has been verified and validated. No inconsistencies were found.

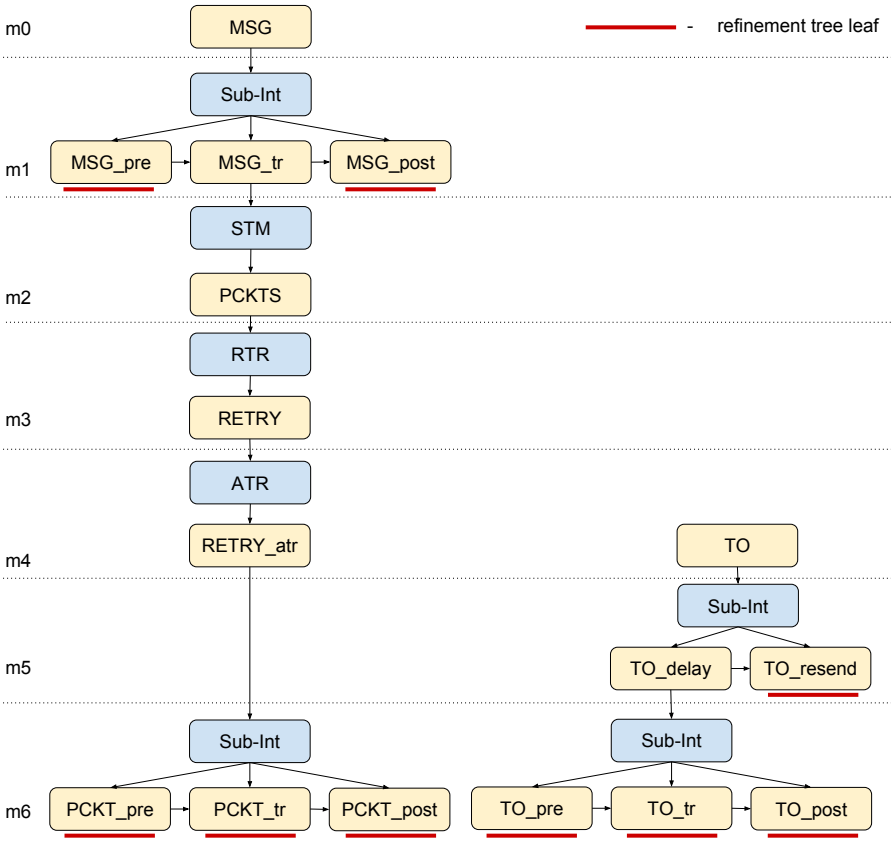

Figure 10.16: Refinement tree of six refinement levels.

Chapter 11

Landing Gear Case Study

We further validate the applicability of our timing interval and its refinement transformations in the third case study. Firstly, we explain the problem domain and define the scope of requirements. Then, we give details on the formal model and the use of our timing interval approach. Finally, we overview the results and conclude.

11.1 Landing Gear System

An aircraft landing gear system case study was proposed by [40] as a benchmark for techniques and tools dedicated to the verification of behavioural properties in systems.

The landing gear system performs two main system operations – the extension and retraction of the landing gears. The system encompasses physical, digital and pilot interface elements. The physical element consists of sensors and a hydraulic system that actuate doors and gears. The hydraulic system is manipulated via electro-valves (EVs). The digital element monitors the system via sensors and controls it accordingly by sending electric signals to the EVs. The interface in the cockpit provides the pilot with the system status. The pilot can initiate gear extension and retraction operations with a handle by switching it down or up respectively.

This case study has several advantages in respect of demonstrating the practicality of our approach. The extension and retraction operations are complex sequences of events that are subject to timing constraints. Therefore, the case study requires a good integration of the two. Timing requirements cover interruptible operations, multiple triggers and responses, aggregate timing constraints and heavily overloaded events – the features of our timing interval approach. The case study makes use of timing interval refinement and multiple interval instances. We used only delay and deadline timing properties in the case study.

11.2 Requirements

The subset of requirements used in this case study was extracted from the [40] document and categorised into three groups (section D.3): environment assumptions (ENV), and functional (FUN) and time (TIME) requirements.

We abstract away from the physical element and focus on the digital element of the system. Our model operates in normal mode and may fail, entering an unrecoverable error state (ENV-1). Since our focus is on timing aspects, there will be more emphasis on time-related requirements and their implementation. The following requirements are given from a landing gear system controller perspective.

Activated by the digital element, one general EV and four actuating EVs regulate the flow of the hydraulic fluid that opens or closes the doors and extends or retracts the gears. EVs require a continuous signal from the controller in order to remain in an open position (FUN-5). When activated, the general EV supplies pressure in the hydraulic circuit (ENV-3) – only then can actuating EVs operate (FUN-6). *Door open* and *door close* EVs actuate doors (ENV-4, ENV-5). Doors can be actuated only when the gears are in one of the final positions (FUN-10). Analogously, *gears extend* and *gears retract* EVs actuate gears (ENV-6, ENV-7). Gears can be stimulated only when the doors are fully open (FUN-9). Opposite EVs (open / close, retract / extend) cannot be stimulated at the same time (FUN-7, FUN-8).

The landing gear system's doors and gears have sensors to help the digital element monitoring the system state. The *Door open* sensor indicates when the doors are fully open (ENV-8). The *Doors closed* sensor indicates when the doors are fully closed (ENV-9). Gears retracted and gears extended work analogously to door sensors (ENV-10, ENV-11). Landing gear components cannot be in the opposite states at the same time (FUN-11). That means that both door sensors or both gear sensors cannot be *TRUE* at the same time.

We simplify the pilot interface and model only the handle. The handle initiates the landing gear extension operation when in an upward position and retraction operation when in a downward position (ENV-12). The handle can be switched to interrupt the operation at any time (FUN-1). When the operation is finished, the system ignores the handle's position until it is changed (FUN-4).

We model two basic scenarios of the landing gear system: the outgoing sequence and the retraction sequence (FUN-2). The outgoing sequence scenario is as follows (FUN-3):

1. Stimulate the general EV.
2. Stimulate the door opening EV.

3. Once the doors are in open position, stimulate the gear outgoing EV.
4. Once the gears are extended, stop the stimulation of the gear outgoing EV.
5. Stop the stimulation of the door opening EV.
6. Stimulate the door closing EV.
7. Once the doors are closed, stop the stimulation of the door closure EV.
8. Stop stimulating the general EV.

The extension and retraction operations are exactly the opposite and their requirements are analogous. Therefore, we cover only extension operation requirements and implementation details.

The system has a number of general performance requirements. Sequential stimulations of the general EV and of the manoeuvring EV must be separated by at least 200ms (TIME-5). Orders to stop the sequential stimulation of the general EV and of the manoeuvring EV must be separated by at least 1s (TIME-6). Two contrary orders (closure / opening doors, extension / retraction gears) must be separated by at least 100ms (TIME-7).

We model two operation-specific timing requirements (remember, we cover only extension operation details). Firstly, if the landing gear command handle has been pushed DOWN while in a fully retracted position, then the gears will be extended and the doors will be closed no sooner than 10s and less than 15 seconds after the handle has been pushed (TIME-1). Secondly, in case the operation is interrupted and resumed during the process, the delay requirement is removed and the gears are expected to fully extend in less than 15 seconds after the handle has been pushed (TIME-3).

11.3 Modelling process

The landing gear system model consists of one abstract machine and two refinements[1]. Firstly, we implement all non-timing requirements in a series of refinements. For the sequencing part, we heavily rely on the iUML tool to express functional requirements in UML diagrams and then superimpose our timing interval. We then reiterate throughout the whole model and add timing.

The advantage of such an approach is twofold. Firstly, sequencing and timing domain challenges are tackled one at a time. Secondly, timing can be abstracted with the sequencing, and the actual timing constraints can be introduced later on.

[1]The landing gear system Event-B model: http://users.ecs.soton.ac.uk/gs6g10/lgModel.zip

11.3.1 Abstract Machine: Controller Cycle

The system is controlled by two controllers running in parallel (ENV-2). The controllers run in an embedded control system loop (Figure 11.1). State-machine is multi-instances, represented as set ECU.

Each controller can be in one of two states: state *chain* is an abstract chain of controller tasks; state *sleep* represents the state in which both ECUs sleep for some time before processing the task chain again. In case of an error (event *raise_error*), an ECU enters the *error* state and then continuously sends an error signal to the cockpit (event *error*). The other ECU then blocks.

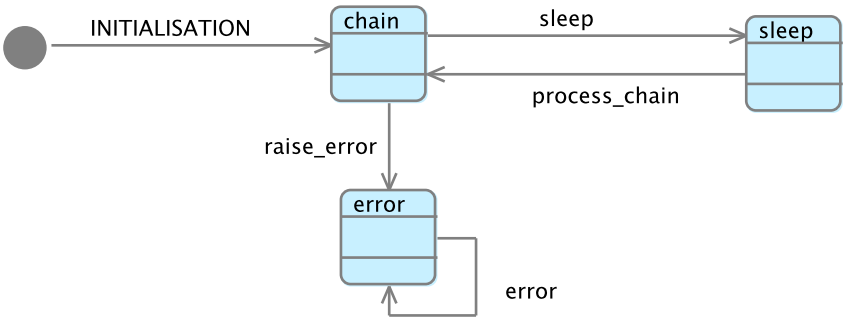

Figure 11.1: Abstract ECU control loop.

$$\begin{aligned} &cycleTI(ECU_2, INIT, sensing\_phase;\\ &\qquad sensing\_phase;\\ &\qquad raise\_error;\\ &\qquad TP_1(DLY, C(CYCLE\_t_{DLY})), TP_2(DDL, C(CYCLE\_t_{DDL}))) \end{aligned} \tag{11.1}$$

The ECU cycle has an execution time-frame and can be aborted by an error.

11.3.2 First Refinement: Process Chain

We elaborate the task chain state with exact operations for the ECUs (11.2). The elaborated ECU loop is based on the failure management system (FMS) [103]. As the title implies, the loop is designed for control systems that deal with faults.

The control chain consists of four stages. The ECU reads data from its sensors in state S. The sensor readings are considered as the inputs to the FMS. The outputs from the FMS are passed to the controller in state C. The task of the FMS is to detect erroneous inputs and prevent their propagation into the controller. Hence the main purpose of the FMS is to supply the controller of the system with the fault-free inputs from the system environment. Both ECUs work in synchrony: they progress the chain tasks together. In

state C, the controller updates its internal model of the system. Finally, in state A the controller performs actuation.

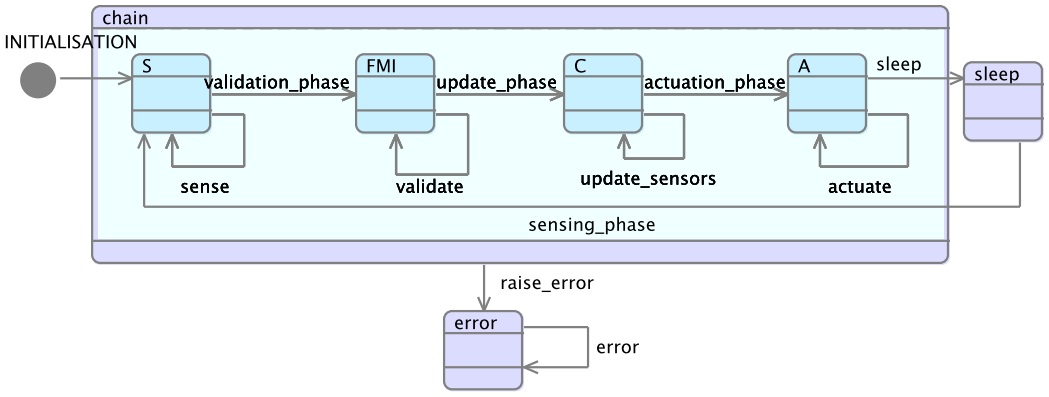

Figure 11.2: Fully detailed ECU control loop.

$$sTI(ECU_2, INIT, sensing\_phase;\ validation\_phase;\ --;\ TP_1(DLY, C(0)), TP_2(DDL, C(0))) \tag{11.2}$$

$$fmiTI(ECU_2, INIT, validation\_phase;\ update\_phase;\ raise\_error;\ TP_1(DLY, C(0)), TP_2(DDL, C(0))) \tag{11.3}$$

$$cTI(ECU_2, INIT, update\_phase;\ actuation\_phase;\ raise\_error;\ TP_1(DLY, C(0)), TP_2(DDL, C(0))) \tag{11.4}$$

$$aTI(ECU_2, INIT, actuation\_phase;\ sleep;\ --;\ TP_1(DLY, C(0)), TP_2(DDL, C(0))) \tag{11.5}$$

$$sleepTI(ECU_2, INIT, sleep;\ sensing\_phase;\ --;\ TP_1(DLY, C(SLEEP\_t_{DLY})), TP_2(DDL, C(SLEEP\_t_{DDL}))) \tag{11.6}$$

We introduce number of parallel state machines that elaborate each stage separately. A sensing task is elaborated with the S state machine (Figure 11.3). While in this state, the ECU can perform sensor-related activity. Note the shared event *sense*. We use the iUML transition property that all outgoing states must be active to synchronise two state machines. The S state machine can operate only when the ECU loop is in the S state. Otherwise the sensing state machine can be aborted by a raised error.

Similarly, state *FMI* is represented by a FMI state machine (Figure 11.4). The control state is represented by state machine (Figure 11.5) for door control. Analogous state machines are created for a gear model, handle model and overall aircraft system model.

We refine the *cycleTI* timing interval into five sub-intervals: (11.2), (11.3), (11.4), (11.5) and (11.6). Each of the sub-intervals corresponds to a state in the state machine in the same order: s, *FMI*, C, A and *sleep*. We make every control loop stage instant and allow time to progress only in state *sleep* by $SLEEP\_t_{DLY,DDL}$ (11.6).

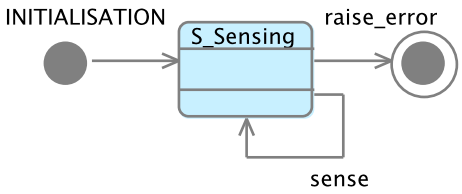

Figure 11.3: Sensing state machine.

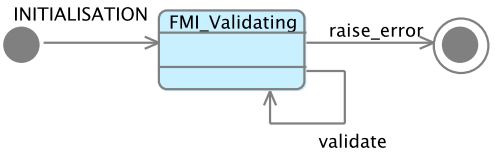

Figure 11.4: FMI state machine.

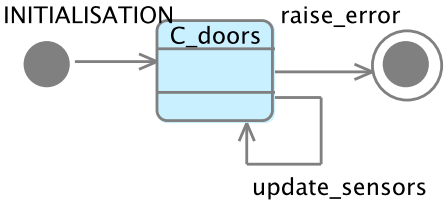

Figure 11.5: Door control state machine.

Note that *fmiTI* and *cTI* timing intervals can be aborted. This is because in these phases an error can occur. In the FMI phase an error can occur due to detected inconsistencies in the sensing data. In the control states, the error may occur if the state fails to update within the given timing constraints.

11.3.3 Second Refinement: Elaborated Control Loop States

We further elaborate sensing-phase sub-state machines (Figure 11.6). While in the sensing phase, the state machine represents the action of reading door sensor input data. As its parent state, the sub-state machine has three instances of it running – one for each triplicated sensor. Each instance can read different readings from the sensor. The door sensor readings can be either *S_DC* – door closed, *s_DO* – door open, or *s_DX* – door in between. The sensor can be declared as faulty by the controller. Such a sensor then enters the *S_Doors_Faulty* state and its readings are ignored. Similarly, we add concurrent state machines for gear and handle sensors. The gear state machine can be in either of states *S_GE* – gear extended, *S_GR* – gear retracted, or *S_GX* – gear in between, while the handle can be in either state *S_HD* – handle down, or state *S_HU* – handle up.

When the sensing phase is complete, the controller enters the fault management phase. We further elaborate the FMI phase for the door, gear and handle sensor validation. The door validation state machine (Figure 11.7) has the following work-flow: upon entering

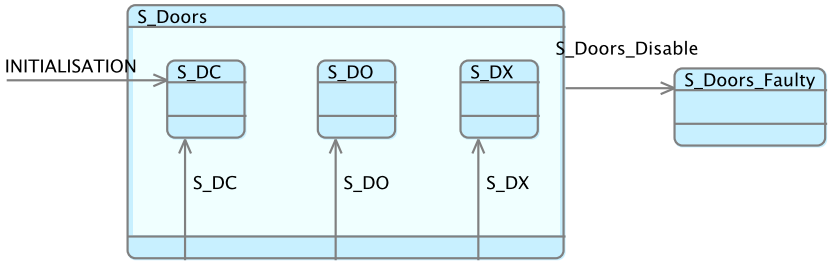

Figure 11.6: Sensing phase: door sensor.

the validation phase, the door FMS starts with the *FMS_Doors* state. If all three sensor readings match, event *S_Doors_OK* sets door state to *FMS_Doors_OK* and eventually the door sensor reading validation successfully passes. In case one of the door readings mismatches, the mismatched sensor is set to faulty state (*S_Doors_Faulty*) and the reading of the other two sensors is taken as the correct one. In case all three (two if one is already disabled) sensors produce different readings, the landing gear system immediately enters error state by executing event *FMS_Doors_Raise_Error*. If the door, gear and handle validations pass successfully, the controller enters the control phase.

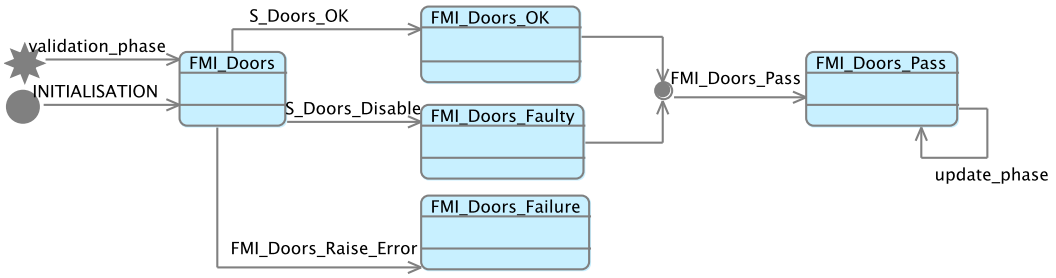

Figure 11.7: FMS phase: door sensor reading validation phase.

We further elaborate the door control phase (Figure 11.8). Depending on the sensor input, the controller then updates the internal model of the landing gear system. It sets the door state to either closed *DC*, open *DO* or in between *DX*. Similarly, we perform the gear control: depending on the sensor data, the gears can be either set to extended, retracted or in between. The handle state machine is analogous.

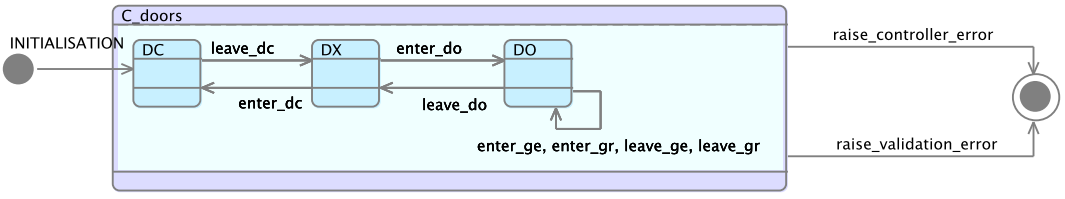

Figure 11.8: Controller state-update phase: internal door state model.

We can determine the overall state of the landing gear system depending on the handle, door and gear sensor readings. We add two states for the state model (Figure 11.9). State *C_final* represents one of the final landing gear system states - either fully open or fully retracted gears with the doors closed. State *C_in_between* represents the system in motion.

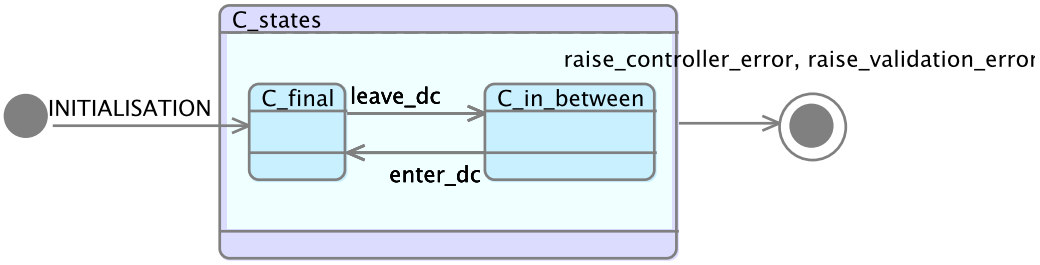

Figure 11.9: Controller state-update phase: internal system model.

We further elaborate the actuation phase by adding the general electro-valve (Figure 11.10).

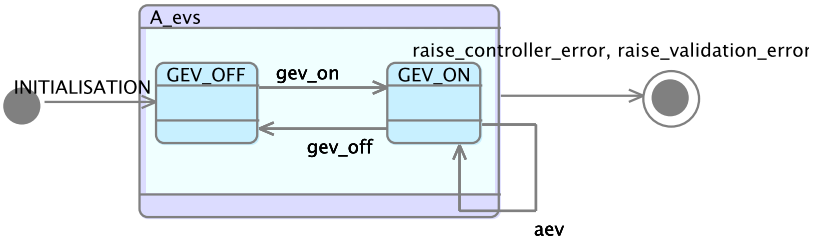

Figure 11.10: Actuation phase: electro-valve states.

$$EVS1(ECU_2, gev\_on;\ aev;\ --;\ TP_1(DLY, C(EVS1\_t_{DLY}))) \tag{11.7}$$

$$EVS2(ECU_2, aev;\ gev\_off;\ --;\ TP_1(DLY, C(EVS2\_t_{DLY}))) \tag{11.8}$$

The ECU has a set of timing constraints that must be respected. Interval R11_R81_R12_R82 (11.9) is the abstract timing requirement that states that from the moment the handle is switched, the system must reach the final state within time $R11\_R81\_R12\_R82\_t_{DDL}$, but no sooner than $R11\_R81\_R12\_R82\_t_{DLY}$. The interval can be aborted by either a changed command (event *H_switch*) or an error.

Interval EVS1 states that after the general electro-valve has been switched on, there has to be a delay of at least time $EVS1\_t_{DLY}$ before other electro-valves can be operated (11.7). Event *aev* represents an abstract event that switches other electro-valves on and off. Interval EVS2 states that the general electro-valve can be switched off only after at least time $EVS2\_t_{DLY}$ has passed after the last manoeuvring electro-valve actuation (11.8).

$$
\begin{aligned}
R11\_R81\_R12\_R82(&ECU_2, H\_switch;\\
&raise\_handle\_timing_error;\\
&H\_switch, raise\_controller\_error, raise\_validation\_error, enter\_dc;\\
&TP_1(DLY, C(R11\_R81\_R12\_R82\_t_{DLY})),\\
&TP_2(DDL, C(R11\_R81\_R12\_R82\_t_{DDL})))
\end{aligned}
\tag{11.9}
$$

11.3.4 Third Refinement

We add four timing intervals in this refinement. Timing interval $D1$ expresses the requirement that when the door-open electro-valve is switched on, either the door must fully open or the valve has to be switched off within time $D1\_t_{DLY,DDL}$, otherwise an error will occur. (11.10). Timing interval $D2$ expresses the analogous requirements for the door close electro-valve (11.11). Timing intervals (11.12) and (11.13) are for gear extend and gear retract electro-valves respectively.

$$
\begin{aligned}
D1(&ECU_2, do\_on;\ raise\_do\_timing\_error;\ enter\_do, do\_off;\\
&TP_1(DLY, C(D1\_t_{DLY})), TP_2(DDL, C(D1\_t_{DDL})))
\end{aligned}
\tag{11.10}
$$

$$
\begin{aligned}
D2(&ECU_2, dc\_on;\ raise\_dc\_timing\_error;\ dc\_off, enter\_dc;\\
&TP_1(DLY, C(D2\_t_{DLY})), TP_2(DDL, C(D2\_t_{DDL})))
\end{aligned}
\tag{11.11}
$$

$$
\begin{aligned}
G1(&ECU_2, ge\_on;\ raise\_ge\_timing\_error;\ ge\_off, enter\_ge;\\
&TP_1(DLY, C(G1\_t_{DLY})), TP_2(DDL, C(G1\_t_{DDL})))
\end{aligned}
\tag{11.12}
$$

$$
\begin{aligned}
G2(&ECU_2, gr\_on;\ raise\_gr\_timing\_error;\ gr\_off, enter\_gr;\\
&TP_1(DLY, C(G2\_t_{DLY})), TP_2(DDL, C(G2\_t_{DDL})))
\end{aligned}
\tag{11.13}
$$

11.3.5 Fourth Refinement

In the fourth refinement, we elaborate the system state model state machine (Figure 11.11).

Accordingly, we elaborate the handle state machine (Figure 11.12). Transitions between states HU and HD with prefix $HU\_$ denote the "switch handle up" action in a specific state, e.g. HU_A denotes the handle-switch action in state A. Self-loop transitions in the states enforce synchronisation. While in state HU, only transitions that relate to the extending action are enabled.

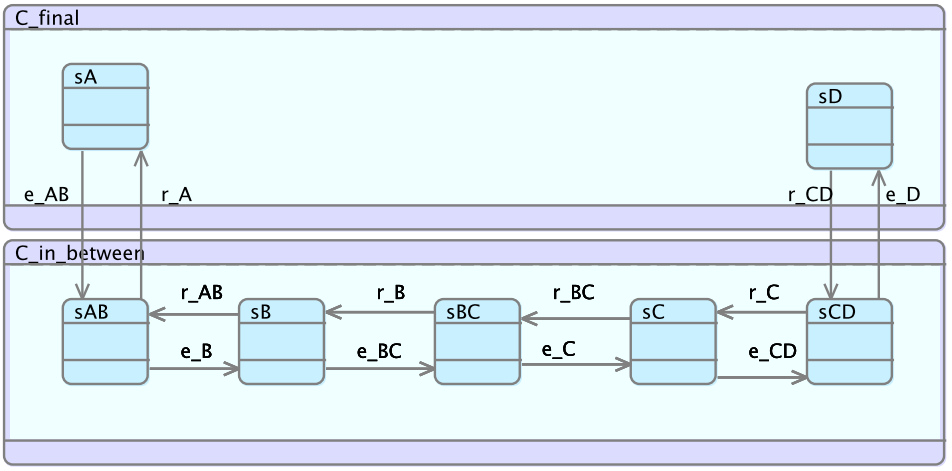

Figure 11.11: Controller state-update phase: internal system model with intermediate and final states.

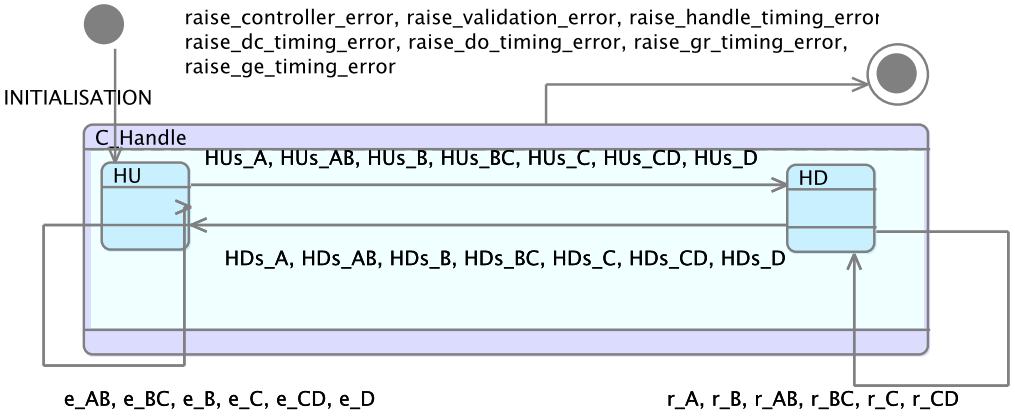

Figure 11.12: Landing gear controller cycle.

11.3.6 Fifth Refinement

In the fifth refinement, we elaborate the system state state machine (Figure 11.13). We add sub-states for every landing gear system position that differentiates activities by the handle being either in the up or down position .

We elaborate the GEV state. We add door the electro-valve state machine (Figure 11.14). The system can stimulate either the door open EV, door close EV or none. The gear-valve state machine is analogous.

We elaborate the handle state machine (Figure 11.15). Handle position determines the end goal for the system. In cases where the handle is switched down, the goal is to fully extend the gears and close the doors. We differentiate handle positions to two states – either the operation is still in progress or complete.

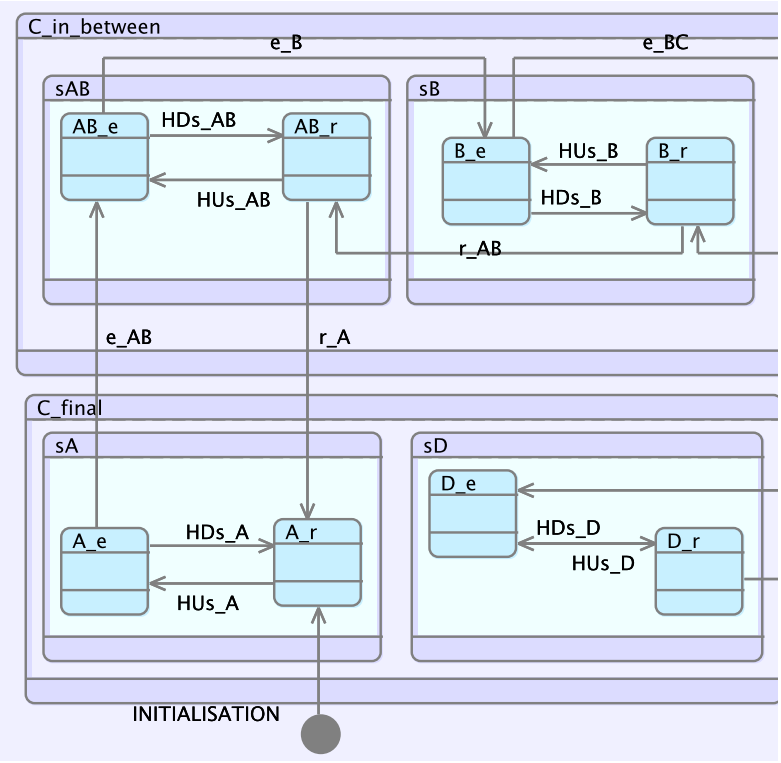

Figure 11.13: Controller state-update phase: elaborated intermediate and final states.

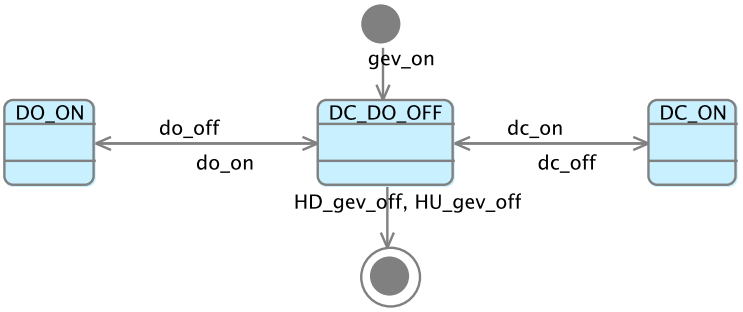

Figure 11.14: Controller actuation phase: door EVs.

In this refinement, we refine timing interval $R11\_R81\_R12\_R82$ into alternative intervals $R11\_R81$ (11.14) and $R12\_R82$ (11.15). The intervals differentiate between the handle up and handle down related dynamics.

11.3.7 Sixth Refinement

In the final refinement, we further elaborate the system state model. The final state (Figure 11.16) is added concrete states for A_r. The initial state A_r_gev is where the model starts. In cases where the handle is switched down, the system enters state $A\_e\_gev\_off$, meaning the GEV is off. The controller can then perform the actions

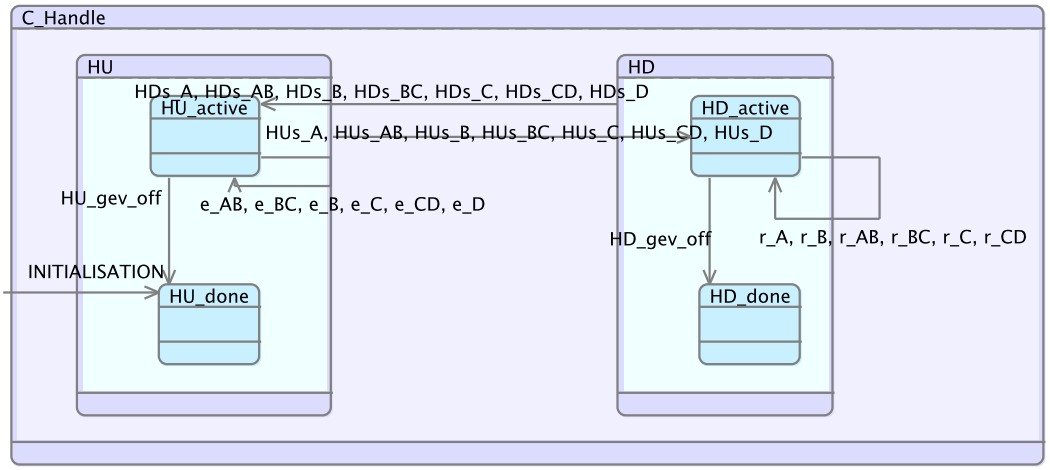

Figure 11.15: Controller state-update phase: elaborated handle states.

$$
\begin{aligned}
&R11\_R81(ECU_2, HDs\_*;\\
&\quad raise\_hd\_timing\_error;\\
&\quad HDs\_*, raise\_controller\_error, raise\_validation\_error, e\_D, r\_A;\\
&\quad TP_1(DLY, C(R11\_R81\_t_{DLY})), TP_2(DDL, C(R11\_R81\_t_{DDL})))
\end{aligned}
\tag{11.14}
$$

$$
\begin{aligned}
&R12\_R82(ECU_2, HUs\_*;\\
&\quad raise\_hu\_timing\_error;\\
&\quad HUs\_*, raise\_controller\_error, raise\_validation\_error, e\_D, r\_A;\\
&\quad TP_1(DLY, C(R12\_R82\_t_{DLY})), TP_2(DDL, C(R12\_R82\_t_{DDL})))
\end{aligned}
\tag{11.15}
$$

listed in the self-loop transition: switch on the GEV $A\_e\_gev\_on$ and actuate the DO EV – $A\_e\_do\_on$. When the DO EV is switched on, the controller updates the system state to $A\_e\_do\_on$, meaning that soon the doors will enter DX state.

At any point in time, the handle can be switched down. The system then enters state $A\_r\_dc\_off$. While in this state, the controller must switch off the DO EV. Then, the controller transitions to state A_r_gev and switches off the GEV to reach the final state.

We show the elaborated AB and B states (Figure 11.17). In the case when the handle remains in the down position and the doors start to open, the system state is updated to $AB\_e\_do\_on$. The latter denotes that the doors are in the between position and the DO EV is on. The system can then remain in this state until the doors fully open and the system enters state $B\_e\_do\_on$. The controller then knows that it is time to switch on the GE EV and the controller eventually enters state e_BC.

Similarly, in case the handle is up and the gears have just been retracted, the system enters state $B\_r\_gr\_on$. This state means that the gears are retracted but the GR EV is still being actuated. The system then transitions to the $B\_r\_gr\_off$ state where the gear actuator can be switched off, then the DC EV is switched on. At this point the system can then progress to state $B\_r\_dc\_on$, meaning that the doors will soon leave the

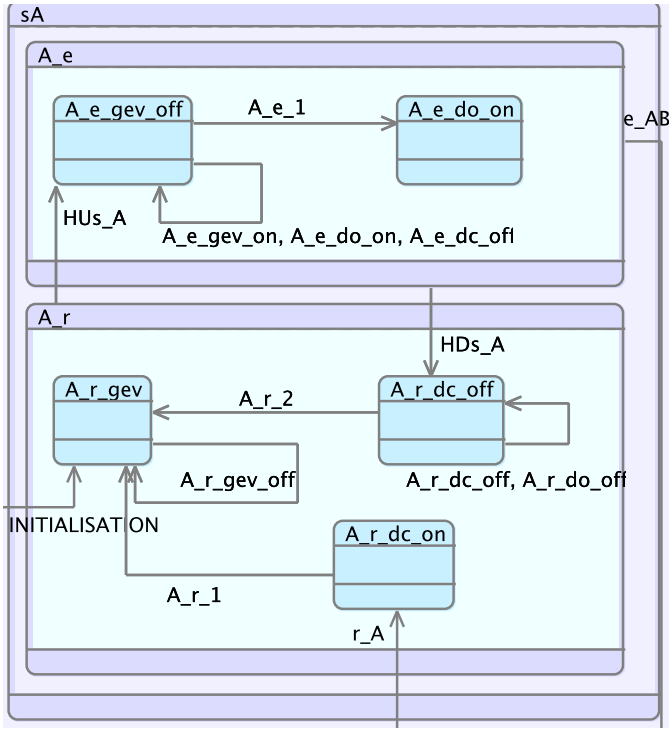

Figure 11.16: Final state A.

DO state. When they do, the system enters state *AB_r_DC_on*. If the handle position does not change, the doors eventually close and the system enters state *A_r_dc_on* (Figure 11.16).

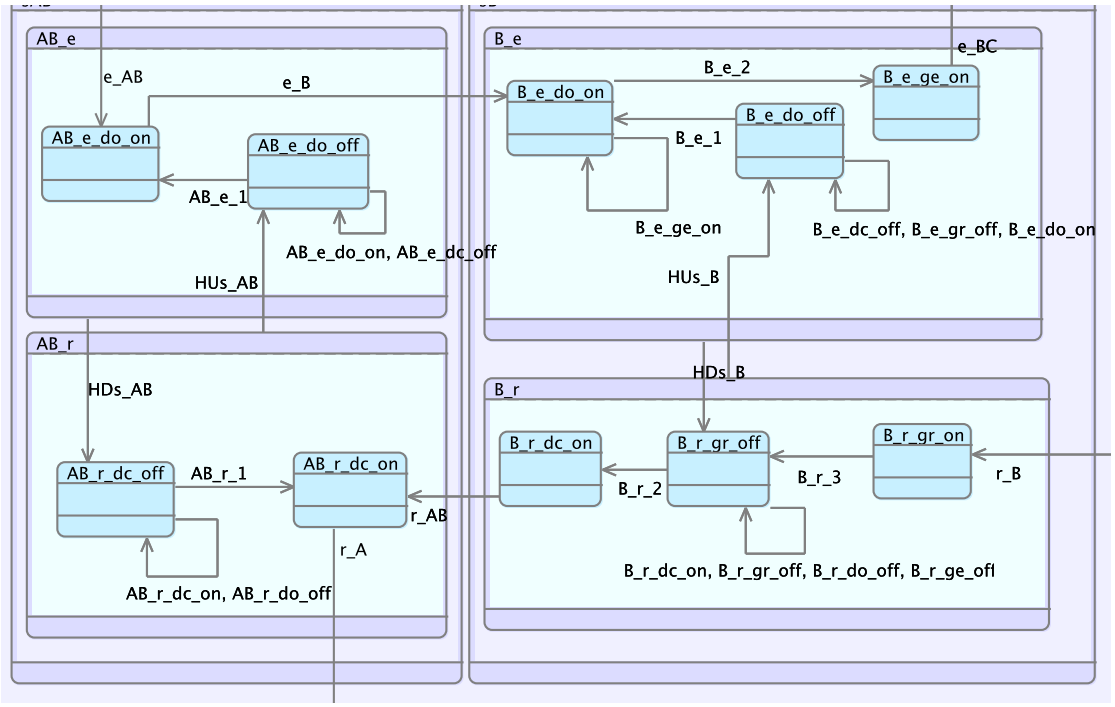

Figure 11.17: States AB and B.

The rest of the system states are modelled in the analogous way.

11.4 Verification and Validation

The final model has resulted in six refinements. A total of 3837 POs were generated, all of which were discharged automatically using our fine-tuned auto prover (section 8.1). The final model has 192 variables, 78 invariants and 122 events. A total of 242 invariants were generated across all refinement levels.

Ref. Lvl	POs	Manual POs	Variables	Invariants	Events
m0	39	0	9	11	6
m1	167	0	28	27	13
m2	665	0	85	26	43
m3	302	0	115	50	54
m4	391	0	124	26	71
m5	904	0	158	24	73
m6	1367	0	192	78	122
Total	**3837**	**0**	192	242	122

Table 11.1: Proving statistics.

The model has been fully model-checked with small fixed timing interval values. No inconsistencies were found. We have manually validated the model with ProB animator. No inconsistencies were found.

11.5 Overview and Conclusions

We have modelled a simplified version of the landing gear case study. We followed the two-phase approach. Firstly, functional system requirements have been expressed in iUML state machine diagrams. With the only exception being manually added invariants that map main landing gear system phases with EV states. Then the timing dimension has been superimposed on the abstract model. We have used [103] to create a fault management system for embedded control systems.

The final model resulted in eleven timing intervals as displayed in the refinement tree in Figure 11.18. Every timing interval has two instances, one for each ECU. We have used Sub-Interval and Alternative refinement transformations. The model has been successfully verified and validated.

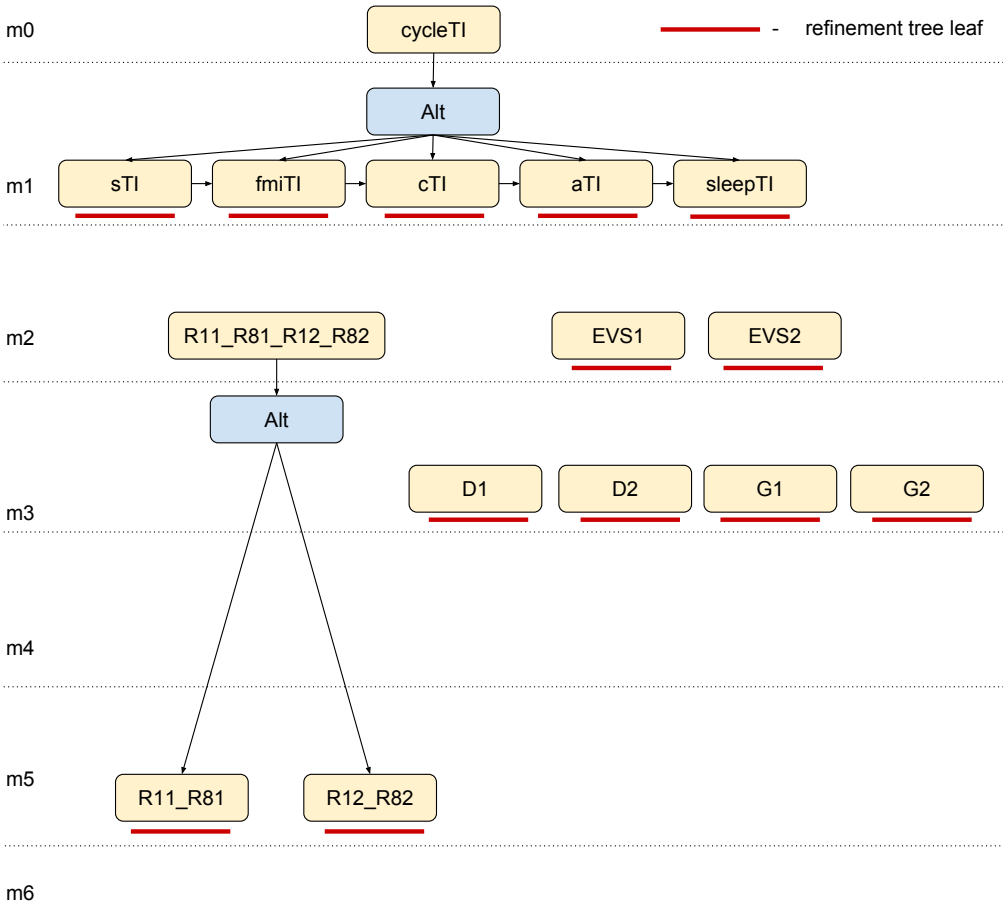

Figure 11.18: Timing interval refinement tree.

Chapter 12

Conclusions and Future Work

In this chapter we summarise our achievements and provide directions for further work.

12.1 Main Contributions

The work in this thesis focuses on developing improved methods of modelling timing requirements in Event-B formal modelling and verification language [19]. We introduce a set of improvements, demonstrate their usefulness and validate them with three case studies.

Our contribution is a methodical scheme with tool support for modelling timing requirements in a refinement setting. This is validated by application to three case studies.

Timing Interval. We address research question RQI with a concept of timing interval (chapter 4) – a higher-level abstraction – in terms of state machine and formal timing interval specification. The timing interval aggregates delay, deadline and expiry timing properties defined by Sarshogh [143]. We present a template-based generative scheme (RQIV) for the transformation of timed models – specified with state machines and timing interval – to Event-B. We demonstrate the generative process and the resulting Event-B semantics in the example model. The timing interval requirements can potentially be fully model-checked due to the use of bounded variables and relative clocks. The timing interval can have multiple instances. The instance can be triggered, responded to and aborted by multiple events, where the event can serve more than one role for many timing intervals. We extended the approach with special cases – singleton interval and dynamic timing property duration. The notion of timing interval lays the ground for the rest of our contribution.

Refinement Transformations. To support the timing interval abstraction through the Event-B based refinement method (RQII) we present five compositional refinement transformations: Sub-Interval, Alternative, Abort-to-Response, Single-to-Multi and Retry (chapter 5). For each of them, we provide generative templates (RQIV) and describe the transformation process and semantics. The given templates take advantage of the generics by reusing existing templates. Multiple refinement transformations can be applied to any abstract interval at any refinement level. We call this feature a parallelised refinement.

Modelling Automation. In chapter 6 we introduce a Rodin plug-in, called tiGen, to automate the timing interval and its refinement modelling process (RQIV). The tool takes advantage of the timing interval and its refinement transformation generative constructs. The tool displays the timing interval and the refinement tree thereof graphically. It works as a stand-alone modelling tool and is compatible with iUML.

Modelling Guidelines. We use iUML state machines and tiGen timing intervals to provide a workflow for modelling timed systems in Event-B (RQIII). We introduce a recommended convention to graphically represent a timing interval and its refinement transformations in state machine diagrams. We provide a method to synchronise state machines and timing intervals formally.

Approach Validation. We have validated our approach and the recommended development workflow (RQIII) in three case studies. Each case study validates an overlapping subset of the schema's features. All models were fully proved and at least partially model checked.

The pacemaker case study (chapter 9) demonstrates how an abstract timing interval can be elaborated through a series of refinements into a complex set of timing intervals. We have used Sub-Interval, Alternative and Abort-to-Response refinement transformations. We have shown the usage of timing intervals with dynamic duration and the parallelised refinement feature. The model has been model-checked with real pacemaker parameters.

The message passing protocol case study (chapter 10) demonstrates how a multi-instance timing interval can be transformed into identical multiple sub-intervals. We have used Single-to-Multi and Retry transformations to transform a message transfer protocol into a concurrent packet transfer process, each having multiple retries. The model has been fully model-checked with one message in transfer and partially model-checked with multiple messages in the transfer.

The landing gear case study (chapter 11) validates multi-instance timing interval with Sub-Interval and Alternative transformations in a synchronised dual-controller system model.

The modelling experience gave us insight into the practicality of the approach. The timing interval facilitated reasoning about the timing requirements by providing a high-level outline of the refinement structure. We have learned that sometimes the existing set of refinement transformations is not enough to transform an abstract timing interval into multiple concrete intervals. For example, a packet transfer time-out requirement in the message passing case study (subsection 10.2.5) was modelled as a separate timing interval and not as a part of the packet transfer itself. Parallel timing intervals are undesirable due to potential deadlocks (discussed in section 7.4). Most of the Event-B specification in the case studies has been generated automatically using iUML and tiGen tools. All time-related proof obligations have been automatically discharged. The case studies have been manually integrated with the iUML state machines, the latter representing the functional aspect in all of the case studies. The automation of the integration process is feasible and would benefit the modelling performance.

The modelling process, especially where parallel and interconnected timing intervals were involved, revealed the challenges in the deadlock verification process. Deadlock detection was quick and fully automated by ProB. Resolving the deadlocks, on the other hand, was time-consuming due to limited event parameter debugging capabilities in ProB. In large models, it was difficult to determine the actual state of a model that caused the deadlock.

We did not find any inconsistencies in the case studies during the modelling process.

12.2 Scope

Our timing interval approach and its tooling support comes with a few limitations. We list the ones that we are aware of.

A timing property constrains all of the timing interval response events. It cannot be applied for a specific response event. We have faced this limitation in the message passing protocol case study (chapter 10).

The timing interval and its refinement transformations are translated to Event-B automatically with a few exceptions. Firstly, the formal mapping between state machines and timing intervals has to be done manually. Secondly, the generation of dynamic duration gluing invariants between the abstract and the concrete refinement levels is not fully automated.

The tiGen tool lacks integration with iUML. Therefore, the process of modelling and expressing timing requirements and their refinements in iUML state machine diagrams (chapter 7) has to be done manually.

12.3 Scalability

In this section we point out two aspects of our approach that may not scale well.

First, we did not investigate the decomposition. Therefore, large models, such as the landing gear case study chapter 11, cannot be split into smaller modules.

Second, our approach never refines away abstract variables. The variables accumulate with every refinement of the timing interval. The only solution to this, that we are of, is to, at some point, perform data refinement of an already built model. We note that here we propose to data refine the resulting timing interval model, which is based on the superposition refinement. We do not propose transforming the timing interval approach itself as discussed in section 5.2. We believe that data refinement of the approach can be methodical due to the use of the template-based approach which produces deterministic Event-B constructs.

12.4 Related Work

Using Event-B to Model time-critical systems has been investigated in several studies. We overview approaches on modelling timing requirements in Event-B in section 3.6 and section 3.7. In the following paragraphs we briefly contrast our work with the work done in Event-B and other formalisms.

Méry et al. [55] and the derived work of Rehm [109] and Bryans et al. [43] does not offer classification of time like Sarshogh [143], and lacks a generative view on modelling time and refinement patterns. We use interval-specific relative clocks, as proposed by [109], rather than a single clock for all intervals to address the infinite state-space problem. In contrast to Berthing et al. [107], our modelling approach is homogeneous in that we rely on Event-B for modelling both functional and timing aspects.

In Sarshogh's work, [143] a timing property is the highest-level element; in our approach, it is part of the aggregate. The latter can then be reasoned upon and refined as a single higher-level abstraction entity. We provide more modelling flexibility by allowing multiple triggers and abort events at the same refinement level and introduce a special kind of response event – abort. [143] refinement patterns are based on data refinement – the refined interval is removed from the model. We rely on the superposition refinement and build concrete timing requirements on the abstract variables. We did not investigate decomposition of our approach and [143] did.

The concepts we use are similar to those in timed automata, which are reviewed in subsection 2.3.1. Among them is the delay, deadline and time-out (expiry) patterns corresponding to our timing properties. The approach proposes a parallel composition pattern, which is similar to our parallel refinement feature regarding functional

behaviour. Timed automata, however, does not support refinement. The latter is the core technique for modelling complex systems in Event-B.

We overview some of the work that has been done to formally verify real-time systems and compare their workflow with ours.

Méry and Singh [71] have developed a single electrode pacemaker system using Event-B. The authors used the activation times pattern [55] to model timing constraints. They did not treat timing constraints as a separate element but rather integrated them tightly into the model. Timing constraint implementation is tightly coupled with the model structure, thus does not take advantage of reusability and requires more modelling effort. Jiang et al. [164] used timed automata to model a closed-loop system of the two-channel pacemaker and the heart. Since UPPAAL lacks a notion of refinement, the complexity of the system is put all at once in a component oriented fashion. The authors modelled pacemaker timing intervals as separate automata that correspond to time counters. The automata communicate via broadcast channels. This is a more complex bottom-up approach than ours. Other works include [101], [25] and [106].

Gomes and Oliveira [25] did a case study of a dual channel pacemaker in Z. Authors designed the pacemaker system in a modular fashion making strong use of the Z schema calculus. Timing aspect in the model is case-specific. For proving, authors used ProofPower-Z theorem prover. They have partially checked the consistency of their specification through reasoning. No validation experiment regarding safety conditions was performed.

Macedo et al. [101] modelled part of the grand challenge pacemaker specification in VDM. They started with a sequential pacemaker model and gradually added concurrency in the later phases. Timing requirements were expressed as timing conjectures: separations, required separations and deadlines. The separation timing conjecture is analogous to our notion of delay, whereas the required separation – to the delay and deadline combination. Conjectures work like assertions and are checked against the traces derived from scenario executions. Authors used test scenarios to model interesting situations such as the absence of input pulses. Similarly, we have defined test case scenarios in our pacemaker case study chapter 9.

None of the reviewed case studies uses notation specific to timing requirements. Modelling of time is case-specific and, with the exception of timed automata based methods, is not represented visually. We are not aware of work suggesting a notion similar to the timing interval.

12.5 Related Methodologies

In this section we compare our work with some of the existing notations and tools that explicitly implement temporal logic.

In terms of suitability for real-time systems, UPPAAL, TLA+ [121] and CSP [144] use real time clocks suitable for continuous systems. Due to the Event-B limitation, we use natural numbers instead, adjusting the precision by changing the interpretation of time units. Real numbers can be added to Event-B via the Theory plug-in [50], but are not yet supported by the ProB model-checker.

In UPPAAL the complexity of the system is managed in a component oriented fashion. The model is then model-checked. In contrast, Event-B, TLA+ and CSP take advantage of refinement to tackle complexity in a stepwise fashion. The models can then be proved and model-checked. Our approach is designed to take advantage of finite state model-checkers. While other approaches [cite] use unbounded clock variables that limit the state-space coverage, we use relative clocks for a potential full state-space model-checking.

Time classification to delay, deadline and expiry are found in many formal methods for real-time systems, sometimes with different names. The Time-Out pattern in timed automata and CSP presents the same property as expiry does in our work and a delay in Timed CSP causes a similar behaviour to what can be modelled by combining a delay and a deadline in our work. In addition to these, timed automata, reviewed in subsection 2.3.1, presents Parallel Composition pattern, similar to our parallel refinement transformation. Timed interrupt pattern, presented in timed automata and Timed CSP, can be reproduced in our work as a sequence of sub-intervals.

Only tiGen and UPPAAL provide a graphical representation of the timing part. UPPAAL displays functional and timing aspects in detail. Our tool focuses on displaying timing interval refinement hierarchy and provides only a high-level overview of the timing interval – events involved (trigger, response, abort), timing properties and their durations.

Temporal properties can be specified and verified in all aforementioned formalisms. For Event-B, a ProB model-checker can be used to specify temporal properties in a form of LTL.

Except for the timing interval, none of the approaches standardise the concept of abort, when an active timing constraint may be abruptly aborted by a non-determinsitic source such as environment.

We are not aware of an approach, except that of Sarshogh's, that presents a timing requirements notation that maps to the underlying formalism semantics and provides a systematic pattern-based approach for refining timing requirements.

12.6 Automatic Gear Change Case Study

Sarshogh modelled an automatic gear change case study based on the [127] specification. In this section, we discuss at a high level the extent to which an automatic gear change case study could be treated with the timing interval approach.

We provide a high-level overview of the timing interval refinement structure for the gearbox change case study (Figure 12.1). All timing intervals displayed in the diagram have only the deadline timing property unless specified otherwise. We abstract away from exact duration values for this high-level overview. The case study and the principal differences between ours and Sarshogh's modelling approaches are explained along the narration of each refinement.

The abstract machine $m0$ and the first refinement $m1$ are similar to their counterparts in Sarshogh's case study. In $m0$ we declare the initial interval *GearChange*, which specifies a generic requirement that the gearbox, upon request, should change the gear within a specified time. The interval is triggered by event *AbstractReq* and responded to by event *AbstractResp*. In the first refinement $m1$, the interval is refined into three possible cases: *FromNeu* - the gear might be changed from neutral to gear, *NoNeu* - from gear to gear, and *ToNeu* - from gear to neutral. All three gear changing cases work on a similar principle. We further cover only the first case. The interval is triggered and responded to with the concrete events *Req* and *Resp* respectively.

In the second refinement $m2$, the response of the abstract interval *FromNeu* is differentiated by the gear-changing process done with or without opening the clutch, events *Cl* and *NoCl* respectively. In Sarshogh's approach, it is required to refine the abstract interval into two alternative response intervals. In our case, it is a simple abstract response event refined into two concrete events. Similarly, the third refinement, $m3$ is concerned only with event specialisation: setting the gear with the clutch – *Setting_Cl* – and without – *Setting_NoCl*. In the fourth refinement $m4$ the interval is refined to *FromNeu_DDL* with a specific deadline duration value.

In the fifth refinement $m5$, we add the intermediate steps required to set the requested gear. In Sarshogh's case study this can be done in a single refinement. The timing interval approach takes three refinement steps – $m5$, $m6$ and $m7$ – to achieve the same result. In $m5$ the abstract interval *FromNeu_DDL* is firstly refined into a sequence of two sub-intervals *Attempt_SS* and *SetGear*. In the first sub-interval, the gearbox attempts to synchronise the gearbox and the engine speeds before setting the gear. Event *ReadyToSet* marks the end of the first sub-interval and triggers the second sub-interval that sets the gear.

In the sixth refinement, event *ReadyToSet* is refined into events *SyncSPeed* and *Open_Cl*, tackling different outcomes. We redraw interval *Attempt_SS* in refinement $m6$ to reflect

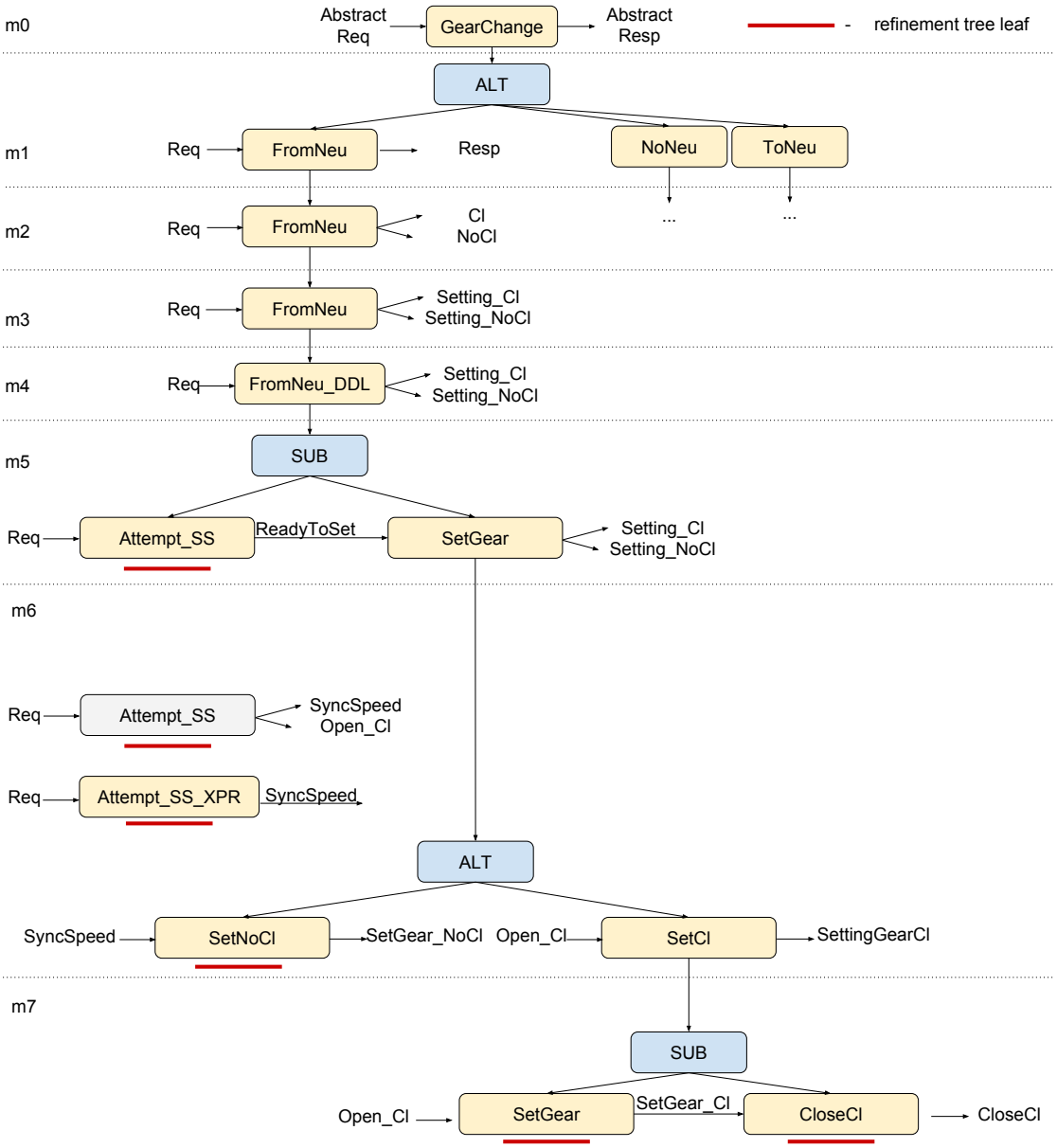

Figure 12.1: Preliminary timing interval refinement tree for the gearbox case study.

the change. The first refining event, *SyncSpeed*, denotes the fact that the gearbox has succeeded in synchronising the speed of the gearbox and the engine. The event triggers interval *SetNoCl*, which sets the gear without the clutch. The task is completed by responding to the interval with event *SetGear_NoCl*. The new timing interval *Attempt_SS_XPR* strengthens the timing requirement for interval *Attempt_SS*. The *Attempt_SS_XPR* interval is constrained only by the expiry timing property with a duration equal to interval *Attempt_SS* deadline. If the gearbox fails to synchronise the speed between the engine and the gearbox while *Attempt_SS* is active, the *SyncSpeed* even blocks and the gearbox is forced to open the clutch in order to change the gear by triggering event *Open_Cl* and starting interval *SetCl*. The latter is responded to by the

event *SettingGearCl*, denoting the fact that the gear change process is complete.

In the final refinement $m7$, interval *SetCl* is refined into two sub-intervals. The first sub-interval *SetGear* sets the gear by triggering its response event *SetGear_Cl*. Sub-interval *CloseCl* then closes the clutch with event *CloseCl*. The latter refines abstract event *SettingGearCl*.

The preliminary refinement structure is similar to that of the Sarshogh's case study. The resulting system can potentially be fully model-checked. The functional part of the system can be modelled manually or by using the iUML tool. The result can then be superimposed with the timing interval refinement structure, which is automatically generated with the tiGen tool. We did not investigate decomposition of the timing interval. Hence further refinements, present in Sarshogh's case study, cannot be reproduced.

12.7 Potential Optimisations of the Tool Chain

In this section, we share our observations about the toolchain and the development process that we have made from the development of the timing interval, the refinement transformations and the three case studies. We point out key areas where the Rodin platform could be improved. The covered aspects are relevant for the industry in order to make the toolchain more appealing in terms of the modelling process, maintenance and efficiency.

Parallelisation. At the moment, the Rodin platform does not take advantage of multi-core CPUs – the proof obligation discharge process is sequential. Similarly, ProB model-checker could parallelise the traversal of a model.

Textual manipulation. Event-B models should be represented and persisted in plain text. Plain text works better with versioning systems such as Subversion [3] or Git [4], as it makes it easier to track changes and compare documents. Plain text can take advantage of the standard text-editing features – such as cut, copy and paste – that improve productivity.

We believe that the modelling style should be enforced with conventions rather than hard-coded in the editor. Depending on a user's experience, the modelling style may change over time, especially when the technology is relatively young and in development.

Graphical modelling. It is difficult to maintain iUML and tiGen diagrams that form a chain of refinement. Editing iUML and tiGen diagrams in the abstract refinement levels, especially by cutting, copying or pasting elements, frequently breaks all of the

concrete diagrams, which are then unrecoverable. The broken diagrams must then be recreated from scratch. We believe that unsupported or unsafe operations should be either disabled or made more robust.

Reusability At present, Event-B does not promote code reuse. Repetitive Event-B constructs reduce the readability of the model and make maintenance difficult. This could be solved by introducing special functions or Event-B event calls from within machine events.

Rodin Development Kit Development of software platforms, such as Rodin, requires a significant amount of resources. In order to make the platform more appealing for contributors, resources should be provided for them to facilitate contribution. We believe that, in terms of software development, focus should be on the development of the Rodin Development Kit – RDK. We suggest this based on industry examples, such as Java Development Kit [5] providing standard libraries for programming in Java; and Google Android Software Development Kit [2] providing resources such as hardware emulators and software libraries for developing software for Android Operating Systems [1].

At the moment, most Rodin plug-ins reimplement the same Event-B modelling logic: the refinement process of a machine, events, variables and the manipulation thereof. Due to limited resources for development and testing, reimplementations lead to increased development overheads, lower quality tools and less focus on research . A Rodin Development Framework would potentially benefit from longer-term support and being more extensively tested. Improvements to the RDK would be reflected in all the tools that depend on it. A good example of code reuse is the core iUML library. The library provides common functionality and building blocks for iUML State-machine, Classdiagram and tiGen plug-ins. From our experience the library significantly reduced the development time of the tiGen tool. However, we had to develop our own code for event and variable manipulation.

12.8 Future Work Directions

There are several interesting directions for further work. This section is split into two parts describing short-term and long-term future work directions.

12.8.1 Short-Term Goals

The short-term future work focuses on the further improvements of the methodology and tool support.

Syntactic Optimisations. We have made the timing interval and its approach generic. The generics provide a greater degree of reusability. However it potentially introduces syntactic redundancy and a greater burden of proof. We believe that the timing interval can be optimised for specific use cases, e.g. when only a single instance of the timing interval is required. This would potentially improve readability and proof of the approach.

More Refinement Transformations. We have learned that sometimes the existing set of refinement transformations is not enough to transform an abstract timing interval into multiple concrete intervals. For example, a packet transfer time-out requirement in the message passing case study (subsection 10.2.5) was modelled as a separate timing interval and not as a part of the packet transfer itself. Parallel timing intervals are undesirable due to potential deadlocks (discussed section 7.4). Therefore, more refinement transformations are required.

Decomposition. Decomposition [48] is a technique to split a system model into component models. The sub-models can then be further elaborated in parallel. We have not investigated the applicability of decomposition in our approach. Decomposition support would allow timed models, such as our message passing protocol, to be split into parallel sub-components, each representing the sender and the receiver. The sub-components could then be further modelled independently.

Sarshogh explains how to decompose a model with shared variables. A shared variable, such as *tick*, firstly is duplicated. Then, the two copies of the variable can be decomposed into separate machines. We believe that a similar principle would work in our approach where most of the variables would have to be duplicated.

Automated Validation. Further research is required regarding verifying deadlock freedom and fairness for timing intervals. Moreover, temporal properties and ProB test cases, such as in the pacemaker case study (section 9.3), could be automatically generated from the timing interval specification. Timing interval and its refinement transformations generate predictable Event-B structures that can potentially be methodically validated and verified. We expect the methodical approach to be more efficient regarding labour and computing resources.

Tool Integration. We have observed that state machines are not expressive enough to unambiguously represent timing interval specifications graphically. More work is required to possibly extend the graphical notation with time interval specific notation. A new plug-in could integrate iUML and tiGen tools by extending them to provide such

graphical extensions and automate Event-B generation of invariants that formally relate and synchronise state machines with timing intervals.

Moreover, the same tool could extend the ProB animator and the iUML state machine animation plug-ins. The tool would co-work with the ProB animator to graphically display timing interval and the dynamics of its variables.

12.8.2 Long-Term Goals

In the long term, our contribution could be used in several areas.

Platform for Temporal Verification Research. The timing interval and tiGen tool could be used as a platform for temporal verification research. The timing interval can be used to construct complex timing constructs such as parallel interconnected timing intervals. The tool can speed up the development and maintenance of such test case scenarios, allowing a researcher to focus on the verification of temporal properties.

Data Refinement and Implementation In this work, we did not consider data refinement and code generation as necessary for the implementation of the actual system. We suggest two research paths to tackle this. First, investigate whether a superposition-based system can be suitable for code generation as is. Second, investigate whether the timing interval can be data refined for code generation. Our system is based on templates, the resulting structure of the timing interval is always deterministic. Therefore a methodical approach is probable.

Case Studies More real-world case studies are needed to highlight the advantages and limitations, and validate and identify the key areas for further improvements of the timing interval.

Heitmeyer et al. have defined a benchmark problem called the Generalized Railroad Crossing [94]. The authors developed a generic version of a real-time railroad crossing system in order to compare different formal methods. The case study could help further evaluate the iUML state-machine and timing interval integration and timing requirements modelling. The results could then be compared to other case studies in terms of applicability. The case study has been modelled in CSP [118], Abstract State Machines (ASM) [88], timed automata [93] and Timed Z [77].

Mikučionis et al. [132] used UPPAAL to perform schedulability analysis of the Herschel-Planck satellite system. Schedulers attempt to serialise parallel-running tasks in such a way that each task would be given a fair amount of computation time to complete before the specified deadline. The case study is interesting in that it could further validate our

approach in terms of modelling parallel timing intervals. The work would serve as a good starting point for temporal verification of multiple timing intervals research as discussed in section 12.8.2. Finally, the case study would contrast timed-automata based modelling against a hybrid approach of Event-B with ProB model-checking capabilities.

Appendix A

Refinement Transformation Templates

A.1 Abort-to-Response

A.1.1 Base Template

Base template call:

$$TPL.ATR.Base(x) \tag{A.1}$$

TEMPLATE **TPL.ATR.Base(x)**

@INVARIANTS

X..i1 : **X_resp** $\subseteq$ **#parent(X)_resp** $\cup$ (**#parent(X)_trig** $\cap$ **#parent(X)_abrt**)

X..i2 : **X_resp_ts** $\in$ **X_resp** $\rightarrow \mathbb{N}$

Figure A.1: ATR: base template.

A.1.2 Transformed Event Template

Template call for abort-response events (Figure A.2):

$$TPL.STM.R\_Transformed(x, atrEvts(x)) \tag{A.2}$$

Where function $atrEvts(x)$ is a set of the transformed events: response events that formerly were abort events.

TEMPLATE **TPL.ATR.R_Transformed**(**x, atrEvts**)

@Event *ALL* ***e*** : ***atrEvts*** $\widehat{=}$

@where

`X..g1` : **#getParam(X, abrt)** = `p_X_resp`

#tc: `TPL.R`(**X**, `e`)

end

Figure A.2: ATR: response event template.

A.2 Single-To-Multi Refinement Transformation

A.2.1 Base Template

Base template for STM (Figure A.3) template call:

$$TPL.STM.Base(x, \#\mathbf{idx(x)}) \tag{A.3}$$

Function **#ran(IDX)** in template returns the range of the provided index set. E.g. **#ran(** $X \times Y$**)** returns X.

TEMPLATE TPL.STM.Base(x, IDX)

#tc: TPL.ROOT.Base(**X**, **#dom(IDX)**)

@INVARIANTS

X..type_i1 : **X**_trig_intr ⊆ **X**_trig

X..consist_i1 : dom(**X**_trig) = **#parent(X)**_trig

X..consist_i2 : dom(**X**_abrt) = **#parent(X)**_abrt

X..consist_i3 : **#parent(X)**_resp = {x | **X**_resp[{x}] = **#ran(IDX)**}

X..consist_i4 : ∀x·x ∈ dom(**X**_trig \ (**X**_resp ∪ **X**_abrt)) ⇔ x ∈ **#parent(X)**_trig \ (**#parent(X)**_resp ∪ **#parent(X)**_abrt)

X..consist_i5 : ∀x, y·x ↦ y ∈ **X**_trig \ (**X**_resp ∪ **X**_abrt) ⇒ **X**_clocks(x ↦ y) = **#parent(X)**_clocks(x)

X..consist_i6 : ∀x, y·x ↦ y ∈ dom(**X**_trig_intr ⩤ **X**_trig_ts) ⇒ **X**_trig_ts(x ↦ y) = **#parent(X)**_trig_ts(x)

Figure A.3: STM: base template.

A.2.2 Delay Template

Delay template call for Single-to-Multi refinement transformation:

$$\forall\, hasDLY(x) = true \Rightarrow TPL.STM.DLY(x, \#\mathbf{idx(x)}, \tag{A.4}$$

$$oldTrigEvts(x), newTrigEvts(x), oldRespEvts(x), newRespEvts(x)) \tag{A.5}$$

Function $oldTrigEvts(x)$ -returns timing interval x trigger events that refine abstract trigger events, $newTrigEvts(x)$ – newly introduced trigger events, $oldRespEvts(x)$ – response events that refine abstract response events, $newRespEvts(x)$ – newly introduces response events.

TEMPLATE TPL.STM.DLY(X, IDX, oldTrigEvts, newTrigEvts, oldRespEvts, newRespEvts)

#tc: TPL.ROOT.DLY(**X**, **oldRespEvts**)

@INVARIANTS

X..i1 : $\forall x, y, z \cdot x \mapsto y \in \mathrm{dom}(\mathbf{X\_trig\_ts}) \land x \mapsto z \in \mathrm{dom}(\mathbf{X\_trig\_ts}) \land y \neq z \Rightarrow (\mathbf{X\_trig\_ts}(x \mapsto y) + \mathbf{X\_t_{DLY}} \leq \mathbf{X\_trig\_ts}(x \mapsto z)) \lor (\mathbf{X\_trig\_ts}(x \mapsto z) + \mathbf{X\_t_{DLY}} \leq \mathbf{X\_trig\_ts}(x \mapsto y))$

X..i2 : $\forall x, y \cdot x \mapsto y \in \mathrm{dom}(\mathbf{X\_trig\_ts}) \land \mathbf{X\_resp}[\{x\}] = \mathbf{\#ran(IDX)} \setminus \{y\} \land x \mapsto y \notin \mathbf{X\_resp} \cup \mathbf{X\_abrt} \Rightarrow \mathbf{X\_trig\_ts}(x \mapsto y) + \mathbf{X\_t_{DLY}} \geq \mathbf{\#parent(X)\_trig\_ts}(x) + \mathbf{X\_t_{DLY}}$

@Event *ALL* ***e*** : ***oldTrigEvts*** $\widehat{=}$
@where
X..g1 : $\mathbf{X\_clocks}(\mathrm{p\_X\_resp}) \geq \mathbf{X\_trig\_ts}(\mathrm{p\_X\_resp}) + \mathbf{X\_t_{DLY}}$
X..g2 : $\mathbf{X\_resp}[\mathrm{dom}(\{\mathrm{p\_X\_trig}\})] \neq \mathbf{\#ran(IDX)} \setminus \mathrm{ran}(\{\mathrm{p\_X\_trig}\})$
end

@Event *ALL* ***e*** : ***newTrigEvts*** $\widehat{=}$
@where
X..g1 : $\forall x, y \cdot x \mapsto y \in \mathrm{dom}(\{\mathrm{p\_X\_trig}\}) \lhd \mathbf{X\_trig} \Rightarrow \mathbf{X\_trig\_ts}(x \mapsto y) + \mathbf{X\_t_{DLY}} \leq \mathrm{p\_X\_trig\_ts}$
end

@Event *ALL* ***e*** : ***newRespEvts*** $\widehat{=}$
@any p_X_dly_ts_aux
@where
X..g1 : $\mathrm{p\_X\_dly\_ts\_aux} \in (\mathbf{X\_trig\_intr} \lhd\!\!+ \mathbf{X\_trig\_ts})[\mathrm{dom}(\{\mathrm{p\_X\_resp}\}) \lhd (\mathbf{X\_trig})]$
X..g2 : $\forall x, y \cdot x \mapsto y \in \mathrm{dom}(\{\mathrm{p\_X\_resp}\}) \lhd \mathrm{dom}(\mathbf{X\_trig\_ts}) \land (\mathbf{X\_resp} \cup \{\mathrm{p\_X\_resp}\})[\{x\}] = \mathbf{\#ran(IDX)} \setminus \{y\} \land x \mapsto y \notin \mathbf{X\_resp} \cup \mathbf{X\_abrt} \Rightarrow \mathbf{X\_trig\_ts}(x \mapsto y) + \mathbf{X\_DLY\_DUR} \geq \mathrm{p\_X\_dly\_ts\_aux} + \mathbf{\#parent(X)\_t_{DLY}}$
end

@Event *ALL* ***e*** : ***oldRespEvts*** $\widehat{=}$
@where
X..g1 : $\mathbf{X\_resp}[\mathrm{dom}(\{\mathrm{p\_X\_resp}\})] = \mathbf{\#ran(IDX)} \setminus \mathrm{ran}(\{\mathrm{p\_X\_resp}\})$
end

Figure A.4: STM: delay template.

A.2.3 Deadline Template

Deadline template call:

$$\forall\, hasDDL(x) = true TPL.STM.DDL(x, newRespEvts(x)) \tag{A.6}$$

Where *newRespEvts* is the list of newly introduced response events.

TEMPLATE **TPL.STM.DDL(X, newTrigEvts)**

#tc: TPL.ROOT.DDL(**X**, **newTrigEvts**)

@INVARIANTS

X..i1 : $\forall \mathtt{idx}, \mathtt{x} \cdot \mathtt{idx} \mapsto \mathtt{x} \in \mathbf{X\_trig} \land \mathtt{idx} \in \mathbf{\#parent(X)\_trig} \Rightarrow \mathbf{\#parent(X)\_trig\_ts}(\mathtt{idx}) + \mathbf{\#parent(X)\_t_{DDL}} \geq \mathbf{X\_trig\_ts}(\mathtt{idx} \mapsto \mathtt{x}) + \mathbf{X\_t_{DDL}}$

@Event *ALL* ***e*** : ***newTrigEvts*** $\widehat{=}$
@where

X..g1 : $\forall \mathtt{x} \cdot \mathtt{x} \in \mathtt{dom}(\mathbf{X\_trig\_intr} \lhd\!\!+ \mathbf{X\_clocks}) \Rightarrow \mathbf{X\_clocks}(\mathtt{x}) - \mathbf{X\_trig\_ts}(\mathtt{x}) + \mathbf{X\_t_{DDL}} \leq \mathbf{\#parent(X)\_t_{DDL}}$

end

Figure A.5: STM: deadline template.

A.2.4 Trigger Templates

Template for trigger events (Figure A.6) called by:

$$TPL.STM.T(x, Tpar(x)) \tag{A.7}$$

Where function $Tpar(x)$ is the set of refined trigger events.

TEMPLATE TPL.STM.T(X, evts)

@Event *ALL* ***e*** : ***evts*** ≙
#tc: TPL.ROOT.T(**X**, **#idx(X)**, **e**, {**#getParam(X, trig_ts)**})
@where

X..g1 : **#getParam(X, trig)** ∈ dom({p_X_trig})

end

Figure A.6: STM: refined trigger event template.

Template for the newly introduced trigger events (Figure A.7) by the following calling template call:

$$TPL.STM.T\_NEW(x, Tnew(x)) \tag{A.8}$$

Where function $Tnew(x)$ is the set of new trigger events.

TEMPLATE TPL.STM.T_NEW(X, evts)

@Event *ALL* ***e*** : ***evts*** ≙
@where
#tc: TPL.ROOT.T(**X**, **#idx(X)**, **e**, (**X_trig_intr** ⩤ **X_clocks**)[dom({p_X_trig}) ◁ (**X_trig** \ (**X_resp** ∪ **X_abrt**))])

X..g1 : dom({p_X_trig}) ⊆ dom(**X_trig** \ (**X_resp** ∪ **X_abrt**))
@then
X..a1 : **X_trig_intr** := **X_trig_intr** ∪ {p_X_trig}
end

Figure A.7: STM: new trigger event template.

A.2.5 Response Templates

Template for the response events (Figure A.8) is called by:

$$TPL.STM.R(x, Rref(x)) \tag{A.9}$$

Where function $Rref(x)$ is the set of refining response events.

TEMPLATE TPL.STM.R(X, evts)

@Event *ALL* ***e*** : ***evts*** $\widehat{=}$
@any p_X_resp
@where

#tc: TPL.ROOT.R(**X**, **e**)
X..g3 : **#getParam(X, resp)** $\in$ dom({p_**X**_resp})
X..g4 : (dom({p_**X**_resp}) $\times$ **#ran(IDX)**) $\subseteq$ **X**_resp $\cup$ {p_**X**_resp}

end

Figure A.8: STM: refined response event template.

Template for the newly introduced response events (Figure A.9) is called by:

$$TPL.STM.R\_NEW(x, Rnew(x)) \tag{A.10}$$

Where $Rnew(x)$ is the set of new response events.

TEMPLATE TPL.STM.R_NEW(X, evts)

@Event *ALL* ***e*** : ***evts*** $\widehat{=}$
@where

#tc: TPL.ROOT.R(**X**, **e**)
X..g1 : (dom({p_**X**_resp}) $\lhd$ **X**_trig) $\setminus$ (**X**_resp $\cup$ **X**_abrt $\cup$ {p_**X**_resp}) $\neq \varnothing$

end

Figure A.9: STM: new response event template.

A.2.6 Abort and Reset Event Templates

Template for abort events (Figure A.10) is called by:

$$TPL.STM.A(x, \#\mathbf{A(x)}) \tag{A.11}$$

Where $\#\mathbf{A(x)}$ is the set of event's abort events.

TEMPLATE **TPL.STM.A(X, evts)**
@Event *ALL e : evts* $\widehat{=}$
@where
X..g1 : p_X_abrt = **#getParam(X, abrt)** $\lhd$ (X_trig $\setminus$ (X_resp $\cup$ X_abrt))
end

Figure A.10: STM: abort event template.

Template for the reset event (Figure A.11) is called by:

$$TPL.STM.RST(x, \#\mathbf{idx(x)}, parRst(x)) \tag{A.12}$$

Where function $parRst(x)$ is the reset event of the parent timing interval.

TEMPLATE **TPL.STM.RST(X, IDX, parRst)**
@Event *parRst* $\widehat{=}$
@any p_X_reset
@where
X..g1 : p_X_reset = p_**#parent(X)**_reset $\times$ **#ran(IDX)**
@then
X..a1 : X_trig := X_trig $\setminus$ p_X_reset
X..a2 : X_trig_ts := p_X_reset $\lhd$ X_trig_ts
X..a3 : X_clocks := p_X_reset $\lhd$ X_clocks
X..a4 : X_resp := X_resp $\setminus$ p_X_reset
X..a5 : X_resp_ts := p_X_reset $\lhd$ X_resp_ts
X..a6 : X_abrt := X_abrt $\setminus$ p_X_reset
X..a7 : X_retry := X_retry $\setminus$ p_X_reset
X..a8 : X_trig_intr := X_trig_intr $\setminus$ p_X_reset
end

Figure A.11: STM: reset event template.

A.3 Retry Refinement Transformation

A.3.1 Base Template

Retry refinement transformation base template (Figure A.12) is called by:

$$TPL.RTR.Base(x, \#\mathbf{idx(x)}) \tag{A.13}$$

TEMPLATE **TPL.RTR.Base(X, IDX)**

#tc: TPL.ROOT.Base(**X**, **#dom(IDX)**)

@INVARIANTS

X..type_i1 : $\mathbf{X}\_retry \subseteq \mathbf{X}\_abrt$

X..type_i2 : $\mathbf{X}\_trig\_intr \subseteq \mathbf{X}\_trig$

X..consist_i1 : $dom(\mathbf{X}\_trig) = I\#\mathbf{parent(X)}\_trig$

X..consist_i2 : $dom(\mathbf{X}\_resp) = \#\mathbf{parent(X)}\_resp$

X..consist_i3 : $\forall x \cdot x \in dom(\mathbf{X}\_abrt \setminus \mathbf{X}\_retry) \Rightarrow x \in I\#\mathbf{parent(X)}\_abrt$

X..consist_i4 : $\forall x \cdot x \in \mathbf{X}\_resp \Rightarrow dom(\{x\}) \not\subseteq dom(\mathbf{X}\_trig \setminus (\mathbf{X}\_resp \cup \mathbf{X}\_abrt))$

X..consist_i5 : $\forall x \cdot x \in dom(\mathbf{X}\_trig \setminus (\mathbf{X}\_resp \cup \mathbf{X}\_abrt)) \Leftrightarrow x \in \#\mathbf{parent(X)}\_trig \setminus (\#\mathbf{parent(X)}\_resp \cup \#\mathbf{parent(X)}\_abrt)$

X..consist_i6 : $\forall x, y \cdot x \mapsto y \in \mathbf{X}\_trig \setminus (\mathbf{X}\_resp \cup \mathbf{X}\_abrt) \Rightarrow \mathbf{X}\_clocks(x \mapsto y) = \#\mathbf{parent(X)}\_clocks(x)$

X..consist_i7 : $\forall x, y \cdot x \mapsto y \in dom(\mathbf{X}\_trig\_intr \lhd \mathbf{X}\_trig\_ts) \Rightarrow \mathbf{X}\_trig\_ts(x \mapsto y) = \#\mathbf{parent(X)}\_trig\_ts(x)$

Figure A.12: RTR: base template.

A.3.2 Event Templates

Abort-trigger template (Figure A.13) is called by:

$$TPL.RTR.AT(x, atEvts) \tag{A.14}$$

Where function $atEvts(x)$ is a list of abort-trigger events.

TEMPLATE **TPL.RTR.AT(X, arEvts)**

#tc: TPL.STM.T_New(**X**, **arEvts**)

#tc: TPL.ROOT.A(**X**)

@Event *ALL* ***e*** : ***arEvts*** $\widehat{=}$

@where

X..g1 : dom(p_**X**_abrt) = dom({p_**X**_trig})

@then

X..a1 : **X**_retry := **X**_retry $\cup$ p_**X**_abrt

end

Figure A.13: RTR: abort-response event template.

Response event template (Figure A.14) is called by:

$$TPL.RTR.R(x, \#\mathbf{R(x)}) \tag{A.15}$$

TEMPLATE **TPL.RTR.R(x, evts)**

@Event *ALL* ***e*** : ***evts*** $\widehat{=}$

@where

#tc: TPL.ROOT.R(**X**, **evts**)

X..g1 : dom({p_**X**_resp}) $\lhd$ **X**_trig $\setminus$ (**X**_resp $\cup$ **X**_abrt $\cup$ {p_**X**_resp}) = $\varnothing$

end

Figure A.14: RTR: response event template.

A.3.3 Deadline Template

The deadline template is identical to the Single-to-Multi (subsection A.2.3). The template call is adjusted for Retry transformation:

$$\forall\, hasDDL(x) = true TPL.STM.DDL(x, newRespEvts(\varnothing)) \tag{A.16}$$

Where function call $newRespEvts(\varnothing)$ returns an empty set. This means that no Event-B will be generated for the events while processing this template. The template generates only invariants.

A.3.4 Delay Template

The delay template is identical to the Single-to-Multi (subsection A.2.3). The template call is adapted:

$$\forall\, hasDLY(x) = true \Rightarrow TPL.STM.DLY(x, \#\mathbf{idx(x)}, \tag{A.17}$$

$$oldTrigEvts(\varnothing), newTrigEvts(x), oldRespEvts(x), newRespEvts(\varnothing)) \tag{A.18}$$

Where function calls with an empty set returns empty set, e.g. $newRespEvts(\varnothing) = \varnothing$. During the generation process events iterating over the empty set list will not generate anything.

Appendix B

Automatic Prover Tactics Configuration

The following XML fragment is the exprot of Auto Prover tactics for Rodin platform, used throughout the three case studies presented in this work. In order to be able to use the provided profile Roding has to have installed Atelier-B and Metaprover plug-ins.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<tactic_pref>
<pref_unit pref_key="Timing Interval Optimised">
<combined combinator_id="org.eventb.core.seqprover.loopOnAllPending"
          tactic_id="org.eventb.core.seqprover.loopOnAllPending.combined">
<simple tactic_id="org.eventb.core.seqprover.partitionRewriteTac"/>
<simple tactic_id="org.eventb.core.seqprover.forallGoalTac"/>
<simple tactic_id="org.eventb.core.seqprover.impGoalTac"/>
<simple tactic_id="org.eventb.core.seqprover.eqvGoalTac"/>
<simple tactic_id="org.eventb.core.seqprover.rmiGoalTac"/>
<simple tactic_id="org.eventb.core.seqprover.lasso"/>
<simple tactic_id="org.eventb.smt.core.autoTactic"/>
<simple tactic_id="com.clearsy.atelierb.provers.core.ml"/>
<simple tactic_id="ch.ethz.eventb.relevancefilter.core.metaprover"/>
</combined>
</pref_unit>
</tactic_pref>
```

Appendix C

Pacemaker Test Case Scenarios

Abbreviations, used in test case scenarios:

- AS – atrial sense.
- AP – atrial pace.
- AX – any atrial activity.
- VS – ventricular sense.
- VP – ventricular pace.
- VX – any ventricular activity.

Test case scenario table fields explained:

- Interval Duration Constants – interval duration values, used for the scenario.
- Initial Active Intervals – intervals, that are already active in the beginning of the scenario. Intervals must not have progressed.
- Scenario Conditions – A list of scenario conditions of what must happen.
- Expected Scenario Result – Conditions at the end of the event execution sequence
- Event Sequence – The sequence of Event-B events in ProB animator to test the scenario.

Test case scenario purpose:

- Atrial sense after ventricular activity – test starts after any ventricular activity. Tests atrial pacing event (a_sense) enablement after executing specified steps.

- Atrial pace after ventricular activity – test starts after any ventricular activity. Tests atrial pacing event (*a_pace*) enablement after executing specified steps.
- Ventricular sense after atrial sense activity – test starts after sensed atrial activity. Tests ventricular sensing event (*v_sense*) enablement after executing specified steps.
- Ventricular sense after atrial sense activity (not during VSP) – test starts after sensed atrial activity. Tests ventricular pacing event (*v_pace*) enablement after executing specified steps.
- No ventricular pace after VSP ends (no VS during VSP) – test starts after paced atrial activity. Tests VSP interval when no intrinsic ventricular activity has been detected ($vsp_e nded$).
- Ventricular pace after VSP ends – test starts after paced atrial activity. Tests VSP interval when intrinsic ventricular activity has been detected ($vsp_e nded_p ace$).

Words in italics are Event-B event names, taken from the pacemaker case study.

Atrial sense after ventricular activity	
Interval Duration Constant	ti_LRI=8; ti_AVI=4; ti_PVARP=3; ti_TARP=7; ti_VAI=4; ti_VRP=2; ti_VSP=2
Initial Active Intervals	LRI; PVARP; VAI; VRP; PVABP; VBP
Scenario Conditions	no VS activity
Expected Scenario Result	a_sense is enabled 3 time units from the initialisation (end of PVARP)
Event Sequence	pvabp_ended; vbp_ended; tick; tick; vrp_ended;tick; pvarp_ended;
Atrial pace after ventricular activity	
Interval Duration Constant	ti_LRI=8; ti_AVI=4; ti_PVARP=3; ti_TARP=7; ti_VAI=4; ti_VRP=2; ti_VSP=2
Initial Active Intervals	LRI; PVARP; VAI; VRP; PVABP; VBP
Scenario Conditions	no VS activity no AS activity
Expected Scenario Result	a_pace is enabled 4 time units from the initialisation (end of VAI) tick event is disabled (due to deadline on VAI)
Event Sequence	pvabp_ended; vbp_ended; tick; tick; vrp_ended;tick; pvarp_ended; tick;
Ventricular sense after atrial sense activity	
Interval Duration Constant	ti_LRI=8; ti_AVI=4; ti_PVARP=3; ti_TARP=7; ti_VAI=4; ti_VRP=2; ti_VSP=2
Initial Active Intervals	AVI; ABP; PAVBP
Scenario Conditions	
Expected Scenario Result	v_sense is enabled 2 time units from the initialisation (when VRP ends)
Event Sequence	pvabp_ended; vbp_ended; tick; tick; vrp_ended;

Ventricular pace after atrial sense activity	
Interval Duration Constant	ti_LRI=8; ti_AVI=4; ti_PVARP=3; ti_TARP=7; ti_VAI=4; ti_VRP=2; ti_VSP=2
Initial Active Intervals	AVI; ABP; PAVBP
Scenario Conditions	no VS
Expected Scenario Result	v_sense is enabled 4 time units from the initialisation (when AVI ends) tick event is disabled (due to AVI deadline)
Event Sequence	pvabp_ended; vbp_ended; tick; tick; vrp_ended;
Ventricular sense (not during VSP) after atrial pace activity	
Interval Duration Constant	ti_LRI=8; ti_AVI=4; ti_PVARP=3; ti_TARP=7; ti_VAI=4; ti_VRP=2; ti_VSP=2
Initial Active Intervals	AVI; ABP; PAVBP; VSP
Scenario Conditions	No VS during VSP
Expected Scenario Result	v_sense is enabled 2 ticks after scenario execution (when VSP ends)
Event Sequence	pavbp_ended; tick; abp_ended; tick; vsp_ended; tick; tick;
Ventricular pace (not during VSP) after atrial pace activity	
Interval Duration Constant	ti_LRI=8; ti_AVI=4; ti_PVARP=3; ti_TARP=7; ti_VAI=4; ti_VRP=2; ti_VSP=2
Initial Active Intervals	AVI; ABP; PAVBP
Scenario Conditions	no VS activity during VSP
Expected Scenario Result	v_pace is enabled 4 time units from the initialisation (when AVI ends) tick is disabled (due to deadline on AVI)
Event Sequence	pavbp_ended; tick; abp_ended; tick; vsp_ended;

Ventricular pace after VSP ends	
Interval Duration Constant	ti_LRI=8; ti_AVI=4; ti_PVARP=3; ti_TARP=7; ti_VAI=4; ti_VRP=2; ti_VSP=2
Initial Active Intervals	AVI; ABP; PAVBP; VSP
Scenario Conditions	VS occurrs during VSP
Expected Scenario Result	vsp_ended is disabled at the end of VSP vsp_ended_pace is enabled at the end of VSP tick event is disabled 2 time units after the scenario execution (due to VSP deadline)
Event Sequence	pavbp_ended; vsp_sense; tick; abp_ended; tick; vsp_ended_pace
No ventricular pace after VSP ends (no VS during VSP)	
Interval Duration Constant	ti_LRI=8; ti_AVI=4; ti_PVARP=3; ti_TARP=7; ti_VAI=4; ti_VRP=2; ti_VSP=2
Initial Active Intervals	LRI; PVARP; VAI; VRP (not progressed)
Scenario Conditions	no VS during VSP
Expected Scenario Result	vsp_ended is enabled at the end of VSP vsp_ended_pace is disabled at the end of VSP tick event is disabled 2 time units after the scenario execution (due to VSP deadline)
Event Sequence	lpavbp_ended; tick; abp_ended; tick; vsp_ended

Appendix D

Case Study Requirement Documents

D.1 Pacemaker DDD Mode Requirements

The compiled requirements document contains one environment (ENV) requirement, one functional requirement (FUN) and 51 timing (TIME) requirements. In this requirement document we do not provide explicit interval durations. The durations are marked with a prefix *ti_*.

The basic assumption is that the pacemaker interacts with the environment (the heart) by sensing and pacing it.

The pacemaker interacts with the heart only by pacing and sensing it	ENV-1

In a healthy heart, atrium and ventricle contractions alternate. The DDD pacemaker must adhere to this behaviour, thus atrial pace (AP) is allowed only after any ventricular activity (VX) and ventricular pace (VP) is allowed only after any atrial activity (AX). Atrial sense (AS) and ventricular sense (VS) are not restricted.

AP is allowed only after VX and VP is allowed only after AX	FUN-1

D.1.1 The Four Fundamental Timing Intervals

The four timing intervals of the DDD mode – atrioventricular interval (AVI), post-ventricular atrial refractory period (PVARP), ventricular refractory period (VRP) and lower rate interval (LRI) – define the whole pacing principle whereas further intervals just specify it in more detail.

D.1.2 Lower Rate Interval

LRI is the longest allowed interval of *ti_LRI* time units between two consecutive ventricular activities.

LRI duration is *ti_LRI* time units	TIME - 1

If no intrinsic ventricular activity occurs during this interval, then the pacemaker delivers a ventricular stimulation at the end of it.

Deliver VP at the end of LRI	TIME - 2

Active LRI an be aborted by VS.

LRI is aborted by VS	TIME - 3

The interval restarts after any ventricular activity.

LRI is triggered by VX	TIME - 4

D.1.3 Ventricular Refractory Period

VRP is defined as a period during which the ventricular channel is insensitive to any incoming signals.

While VRP is active, no VX is allowed	TIME - 5

VRP is initiated by any ventricular activity and lasts for *ti_VRP* time units.

VRP is initiated by VX activity	TIME - 6

The interval can be aborted by intrinsic atrial activity.

VRP can be aborted by AS activity	TIME - 7

VRP duration is *ti_VRP* time units	TIME - 8

VRP is shorter than LRI.

$ti\_VRP < ti\_VAI$	TIME - 9

D.1.4 Atrioventricular Interval

AVI is designed to maintain atrioventricular (AV) synchrony - the interval between contiguous atrial and ventricular events. The interval is initiated by any atrium activity.

AVI is initiated by AX	TIME - 10

If no ventricular activity is sensed during AVI, ventricular stimulation is delivered at the end of the interval.

Deliver VP at the end of AVI	TIME - 11

While AVI is active, interval may not be reset by any atrial activity, and may be only interrupted by any ventricular activity [35].

While AVI is active, no AX is allowed	TIME - 12

At any time, interval AVI can be aborted by intrinsic ventricular activity.

VS can interrupt AVI	TIME - 13

AVI lasts for ti_AVI time units and must be shorter than ti_LRI.

AVI duration is ti_AVI time units	TIME - 14

$ti\_AVI < ti\_LRI$	TIME - 15

D.1.4.1 pAVI and sAVI

Interval AVI can be further classified into atrial pace-triggered or sense-triggered intervals $pAVI$ or $sAVI$ respectively. The intervals may differ in their durations that we define as ti_sAVI and ti_pAVI.

$ti\_sAVI < ti\_AVI$	TIME - 16

$ti\_pAVI < ti\_VI$	TIME - 17

D.1.5 Postventricular Atrial Refractory Period

PVARP is designed to prevent atrial channel from sensing ventricular stimulus,. It is not allowed to initiate a new AVI during active PVARP [35].

While PVARP is active, no AX is allowed	TIME - 18

PVARP is initiated by any ventricular activity and lasts for ti_PVARP time units, but must be shorter than ti_LRI.

PVARP is initiated by VX	TIME - 19

The interval can be aborted at any time by intrinsic ventricular activity.

PVARP can be aborted by VS activity	TIME - 20

Interval's duration ti_PVARP is shorter than VAI.

PVARP duration is ti_PVARP time units	TIME - 21

$ti\_PVARP < ti\_VAI$	TIME - 22

Sometimes ventricular activity can cause cross-sensing. In order to prevent this, PVARP should be always longer than the programmed VRP [152].

$ti\_PVARP \leq ti\_VAI$	TIME - 23

D.1.6 Derived Requirements

The first derived interval is **total atrial refractory period (TARP)**. TARP duration is a sum of all atrial refractory periods, hence $ti\_TARP = ti\_AV + ti\_PVARP$, where ti_AV is the actual duration of the AVI, which can be either $ti\_AV = ti\_AVI$ in the absence of intrinsic ventricular activity or $ti\_AV < ti\_AVI$ in case of sensed ventricular activity.

$ti\_TARP = ti\_AVI + ti\_PVARP$	TIME - 24

TARP must be shorter than ti_LRI.

$ti\_TARP < ti\_LRI$	TIME - 25

The second derived interval is called **upper rate interval (URI)**. The interval is initiated by any ventricular activity.

URI is reset by VX	TIME - 26

While URI is active, pacemaker can not deliver ventricular stimuli.

No VP while URI is active	TIME - 27

The third derived interval is a **ventriculoatrial interval (VAI)**, also known as atrial escape interval (AEI). Interval represents the maximum possible delay for the atrial activity to occur after the ventricle activity. The interval is initiated by any ventricular activity [35].

VAI duration is ti_VAI time units	TIME - 28

VAI is initiated by VX	TIME - 29

The sum of ti_VAI and ti_AVI is equal to ti_LRI.

$ti\_VAI + ti\_AVI = ti\_LRI$	TIME - 30

If VAI runs out, the pacemaker must deliver the atrial pace.

Deliver AP at the end of VAI	TIME - 31

In case of atrial activity is detected while VAI is active, the interval is interrupted.

AS and VS can interrupt VAI	TIME - 32

VAI and AVI, and their sub-intervals, do not overlap.

VAI and AVI do not overlap	TIME - 33

D.1.7 Atrial and Ventricular Blanking Periods

Blanking periods help to avoid crosstalk, when one chamber gets paced by the pacemaker and the signal may be sensed in the other chamber. While during refractory periods the pacemaker may still sense signals that may influence certain timing cycles including automatic mode switching (not covered in this document), blanking periods prevent sensing at all [35].

D.1.8 Ventricular Blanking Period

VBP is initiated by paced ventricular activity and overlaps with VRP.

VBP is initiated by VP	TIME - 34

VBP starts with and overlaps VRP	TIME - 35

VBP lasts for ti_VBP time units and is shorter than ti_VRP.

VBP duration is ti_VBP time units	TIME - 36

$ti\_VBP \geq ti\_VRP$	TIME - 37

D.1.9 Postatrial Ventricular Blanking Period

PAVBP is initiated by paced atrial activity and overlaps with ventricular safety period (subsection D.1.12).

PAVBP is initiated by AP	TIME - 38

PAVBP starts with and overlaps VSP	TIME - 39

PVABP duration is ti_PAVBP time units and is shorter than ti_VSP, which is the duration of ventricular safety period duration.

PAVBP duration is ti_PAVBP time units	TIME - 40

$ti\_PAVBP \leq ti\_VSP$	TIME - 41

D.1.10 Atrial Blanking Period

ABP is initiated by paced ventricular activity and overlaps with AVI.

ABP is initiated by AP	TIME - 42

ABP starts with and overlaps ARP	TIME - 43

ABP lasts for *ti_ABP* time units and is shorter than *ti_AVI*.

ABP duration is *ti_ABP* time units	TIME - 44

$ti\_ABP \leq ti\_VSP$	TIME - 45

D.1.11 Postventricular Atrial Blanking Period

PVABP is initiated by paced ventricular activity and overlaps with PVARP.

PVABP is initiated by VP	TIME - 46

PVABP starts with and overlaps PVARP	TIME - 47

PVABP lasts for *ti_PVABP* time units and is shorter than *ti_PVARP*.

PVABP duration is *ti_PVABP* time units	TIME - 48

$ti\_PVABP < ti\_PVARP$	TIME - 49

D.1.12 Ventricular Safety Pacing

Ventricular safety period (VSP) is initiated by paced atrium and therefore resides in the beginning of $pAVI$ interval.

VSP is initiated by AP	TIME - 50

If any activity is sensed (possible crosstalk effect, but impossible to determine) in the ventricular chamber during VSP, ventricular stimulation is delivered at the end of the VSP to ensure that ventricle has been paced.

If VS occurred during VSP, deliver VP at the end of VSP	TIME - 51

If no VS occurred during VSP, end VSP after ti_VSP time units	TIME - 52

VSP duration is ti_VSP time units	TIME - 53

VSP duration ti_VSP must be shorter than ti_AVI.

$ti\_VSP < ti\_AVI$	TIME - 54

D.1.13 Value Ranges for Pacemaker Intervals

Values collected from the following sources: . Values are provided in ms.

The specified interval AVI duration is the minimum permitted duration. As demonstrated in the pacemaker case study, the value may be extended.

- m0:
  - $LRI \in 340..2000$
  - $URI \in 340..1200$
- m1:
  - $AVI \in 30 - 100$
  - $VAI = LRI - AVI$
- m2:

  - $VRP \in 150..500$
  - $VRP\_off = VAI - VRP$
  - $PVARP \in 150..500$
  - $PVARP\_off = VAI - PVARP$

- m3:
  - $pAVI \in 30..100$
  - $sAVI \in 30..100$

- m4:
  - $VSP = 110$
  - $VSP\_off = pAVI - VSP$

- m5:
  - $sVRP \in 20..50$
  - $pVRP \in 20..50$
  - $sPVARP \in 20..50$
  - $sPVARP \in 20..50$

- m6:
  - $VBP \in 20..50$
  - $VPB\_off = VRP - VBP$
  - $PVABP \in 20..50$
  - $PVABP\_off = PVARP - PVABP$
  - $ABP \in 20..50$
  - $ABP\_off = VSP - ABP$
  - $PAVBP \in 20..50$
  - $PAVBP\_off = VSP - PAVBP$

D.1.14 Model-Checked Values

We reduced values by 10 to shorten the modelling duration. To get the actual durations used, multiply the given value by 10.

- m0:
  - $LRI \in 40$

  - $URI \in 34$

- m1:
  - $AVI \in 3$
  - $VAI = LRI - AVI$
- m2:
  - $VRP \in 15$
  - $VRP\_off = VAI - VRP$
  - $PVARP \in 15$
  - $PVARP\_off = VAI - PVARP$
- m3:
  - $pAVI \in 3$
  - $sAVI \in 3$
- m4:
  - $VSP = 11$
  - $VSP\_off = pAVI - VSP$
- m5:
  - $sVRP \in 2$
  - $pVRP \in 2$
  - $sPVARP \in 2$
  - $sPVARP \in 2$
- m6:
  - $VBP \in 2$
  - $VPB\_off = VRP - VBP$
  - $PVABP \in 2$
  - $PVABP\_off = PVARP - PVABP$
  - $ABP \in 2$
  - $ABP\_off = VSP - ABP$
  - $PAVBP \in 2$
  - $PAVBP\_off = VSP - PAVBP$

D.2 Message Passing Protocol Requirements

D.2.1 Environment Assumptions

The sender, the communications link and the receiver support full-duplex communications	ENV-1
A communication link exists to send the data over to the receiver	ENV-2
A receiver exists to receive the message	ENV-3
Packets may be lost by the communications link and the receiver	ENV-4
We assume that the sender may not have faults	ENV-5

D.2.2 Functional Requirements

It should be possible to use the sender to send a message	FUN-1
Message transfer process comprises three stages: 1. pre-processing of the message to be sent out; 2. the actual message transfer; 3. processing received response data.	FUN-2
The message transfer begins with pre-processing	FUN-3
The message transfer process completes after when response packets are processed	FUN-4
A message consists of multiple packet transfer processes	FUN-5
Multiple packet transfers can be active at a time	FUN-6

In case a packet is lost, the whole packet transfer process (all three stages) must be repeated	FUN-7
A packet should not be resent more than *MaxResend* times ($MaxResend > 1$)	FUN-8
If the sender enters an unrecoverable system state, new packet transfers are not allowed, but the existing packet transfers may complete	FUN-9
Packet transfer process consists of a number of packet transfer attempts	FUN-10
Packet transfer attempt comprises three stages: 1. pre-processing of the packet to be sent out; 2. the actual packet transfer; 3. processing received packet response data.	FUN-11
A response packet has to have the same id as its corresponding sent out packet by the sender	FUN-12
The packets can be sent in any order	FUN-13
The packet responses can be received in any order regardless of when they were sent out	FUN-14

D.2.3 Timing Constraints

Message pre-process stage must finish within time $MESSAGE\_pre\_t_{DDL}$.	TIME-1
Message transfer stage must finish within time $MESSAGE\_tr\_t_{DDL}$ time respectively.	TIME-2
Message post-process stage must finish within time $MESSAGE\_post\_t_{DDL}$.	TIME-3
A successful transferring process of a message must complete within time $MESSAGE\_t_{DDL} = MESSAGE\_pre\_t_{DDL} + MESSAGE\_tr\_t_{DDL} + MESSAGE\_post\_t_{DDL}$	TIME-4

Packet pre-process stage must finish within time $PCKT\_pre\_t_{DDL}$.	TIME-5
Packet transfer stage must finish within time $PCKT\_tr\_t_{DDL}$ time respectively.	TIME-6
Packet post-process stage must finish within time $PCKT\_post\_t_{DDL}$.	TIME-7
A successful transferring process of a packet must complete within time $PCKT\_t_{DDL} = PCKT\_pre\_t_{DDL} + PCKT\_tr\_t_{DDL} + PCKT\_post\_t_{DDL}$	TIME-8
A packet transfer process must be repeated within TO_t_{DDL} time-units, if its transfer process has not been completed and no unrecoverable error has happened in the sender	TIME-9
A packet transfer process can be repeated if at least TO_t_{DLY} time-units has passed from the previous attempt	TIME-10
The next packet has to be sent at the same time as the previous response packet is arrived	TIME-11

D.2.4 Error Detection

When the sender has not received the acknowledgement of the last transferring attempt of a packet, within its deadline, the sender enters the error state	ERR-1

D.3 Landing Gear System Requirements

Landing gear system case study is based on requirements, taken from [40] document.

D.3.1 Environment Assumptions

Landing gear system and its environment always work as expected and never fails	ENV-1
Two controllers run in parallel, read input from sensors and actuate electric valves accordingly	ENV-2

General EV supplies the whole landing gear system with hydraulic power	ENV-3
Door Open EV sets pressure on the portion of the hydraulic circuit related to door opening	ENV-4
Door Close EV sets pressure on the portion of the hydraulic circuit related to door closing	ENV-5
Gears Extend EV sets pressure on the portion of the hydraulic circuit related to gear extending	ENV-6
Gears Retract EV sets pressure on the portion of the hydraulic circuit related to gear retracting	ENV-7
Doors Opened sensor indicates when the doors are fully open	ENV-8
Doors Closed sensor indicates when the doors are fully closed	ENV-9
Gears Extended sensor indicates when the gears are fully extended	ENV-10
Gears Retracted sensor indicates when the gears are fully retracted	ENV-11
Handle can be either in Up (gear retraction) or Down (gear extension) position	ENV-12

D.3.2 Functional Requirements

Operation can be interrupted and reversed at any time if other requirements permit	FUN-1
Two main operations are defined: gear extension and gear retraction	FUN-2
Gear extension is achieved in four stages: A. Doors Closed and Gears Retracted; B. Doors Opened and Gears Retracted; C. Doors Opened and Gears Extended; D. Doors Closed and Gears Extended	FUN-3
When the operation (gears extending or retracting) is complete, the handle remains in the same position and is ignored by the system	FUN-4
The controller must send a continuous signal to keep the EV open	FUN-5

It is not possible to stimulate the manoeuvring electro-valve (opening, closure, outgoing or retraction) without stimulating the general electro-valve	FUN-6
Opening and closure doors electro-valves are not stimulated simultaneously	FUN-7
Outgoing and retraction gears electro-valves are not stimulated simultaneously	FUN-8
The stimulation of the gears outgoing or the retraction electro-valves can only happen when the doors are open	FUN-9
The stimulation of the doors opening or closure electro-valves can only happen when the gears are down or up	FUN-10
Landing gear components can not be in opposite states (DO and DC, GE and GR) at the same time	FUN-11

D.3.3 Timing Constraints

If the landing gear command handle has been pushed DOWN while in fully retracted position, then the gears will be extended and the doors will be seen closed no sooner than 10s and less than 15 seconds after the handle has been pushed	TIME-1
If the landing gear command handle has been pushed UP while in fully extended position, then the gears will be retracted and the doors will be seen closed no sooner than 10s and less than 15 seconds after the handle has been pushed	TIME-2
If the landing gear command handle has been pushed DOWN and stays DOWN, then the gears will be extended and the doors will be seen closed less than 15 seconds after the handle has been pushed	TIME-3
If the landing gear command handle has been pushed UP and stays UP, then the gears will be retracted and the doors will be seen closed less than 15 seconds after the handle has been pushed	TIME-4
Stimulations of the general EV and of the manoeuvring EV must be separated by at least 200ms	TIME-5

Orders to stop the stimulation of the general EV and of the manoeuvring EV must be separated by at least 1s	TIME-6

Two contrary orders (closure / opening doors, extension / retraction gears) must be separated by at least 100ms	TIME-7

Appendix E

tiGen Source Code and Event-B Models

E.1 tiGen Source Code

The tiGen source code is available in the following address: https://eprints.soton.ac.uk/413746/2/tigen_plugin_source.zip

E.2 Event-B Model of the Pacemaker Passing Case Study

The model is available in the following address: https://eprints.soton.ac.uk/413746/5/pacemaker_final.zip

E.3 Event-B Model of the Message Passing Protocol Case Study

The model is available in the following address: https://eprints.soton.ac.uk/413746/4/message_passing_final.zip

E.4 Event-B Model of the Landing Gear System Case Study

The model is available in the following address: https://eprints.soton.ac.uk/413746/3/landing_gear_final.zip

References

[1] Android. https://www.android.com/.

[2] Android Software Development Kit. https://developer.android.com/studio/index.html.

[3] Apache Subversion. subversion.apache.org.

[4] Git. https://git-scm.com/.

[5] Java Development Kit. http://www.oracle.com/technetwork/java/javase/documentation/index.html.

[6] Deploy Project. http://www.deploy-project.eu/, 2008 to 2012.

[7] Advance Project. http://www.advance-ict.eu/, 2011 to 2014.

[8] Abstract Solutions xUML. http://www.abstractsolutions.co.uk/XUML/, 2016.

[9] ADA Implementation Language. http://www.adacore.com/adaanswers/about/ada, 2016.

[10] Isabelle/HOL. https://isabelle.in.tum.de/, 2016.

[11] iUML. http://wiki.event-b.org/index.php/IUML-B, 2016.

[12] MARTE. http://www.omg.org/spec/MARTE/, 2016.

[13] Open Modelica. https://www.openmodelica.org/, 2016.

[14] Papyrus. https://eclipse.org/papyrus/, 2016.

[15] PolarSys. https://www.polarsys.org/, 2016.

[16] Stood. https://wiki.sei.cmu.edu/aadl/index.php/Stood, 2016.

[17] UML. http://www.uml.org/, 2016.

[18] Jean-Raymond Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, New York, NY, USA, 1996.

[19] Jean-Raymond Abrial. *Modeling in Event-B: System and Software Engineering.* Cambridge University Press, 2010.

[20] Jean-Raymond Abrial, Michael Butler, Stefan Hallerstede, ThaiSon Hoang, Farhad Mehta, and Laurent Voisin. Rodin: An Open Toolset for Modelling and Reasoning in Event-B. *STTT*, 12(6):447–466, 2009.

[21] V. S. Alagar and K. Periyasamy. The Z Notation. In *Specification of Software Systems*, pages 461–538. Springer London, London, 2011.

[22] Rajeev Alur, Costas Courcoubetis, Thomas A. Henzinger, and Pei Hsin Ho. Hybrid automata: An algorithmic approach to the specification and verification of hybrid systems. In Robert L. Grossman, Anil Nerode, Anders P. Ravn, and Hans Rischel, editors, *Hybrid Systems*, pages 209–229. Springer Berlin Heidelberg, Berlin, Heidelberg, 1993.

[23] Rajeev Alur and David L. Dill. A Theory of Timed Automata. *Theor. Comput. Sci.*, 126(2):183–235, April 1994.

[24] Rajeev Alur and David L Dill. Automata-Theoretic Verification of Real-Time Systems. *Formal Methods for Real-Time Computing*, pages 55–82, 1996.

[25] Artur O. Gomes and MarcelV.M. Oliveira. Formal Development of a Cardiac Pacemaker: From Specification to Code. In *Formal Methods: Foundations and Applications*, volume 6527 of *LNCS*, pages 210–225. Springer, 2011.

[26] Babak Nazer. *Essential Concepts of Electrophysiology Through Case Studies: Intracardiac EGMs.* Cardiotext Publishing, 2016.

[27] R. J. R. Back and J. Von Wright. *Refinement Calculus: A Systematic Introduction.* Graduate Texts in Computer Science. Springer Verlag, 1998.

[28] Ralph-Johan Back and Reino Kurki-Suonio. Decentralization of Process Nets with Centralized Control. In *Symposium on Principles of Distributed Computing*, pages 131–142, Montreal, Quebec, Canada, 1983. ACM.

[29] Frédéric Badeau and Arnaud Amelot. Using B as a High Level Programming Language in an Industrial Project: Roissy VAL. In Helen Treharne, Steve King, Martin Henson, and Steve Schneider, editors, *ZB 2005: Formal Specification and Development in Z and B: 4th International Conference of B and Z Users, Guildford, UK, April 13-15, 2005. Proceedings*, pages 334–354. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005.

[30] Christel Baier, Boudewijn Haverkort, Holger Hermanns, and Joost-Pieter Katoen. Model-checking algorithms for continuous-time Markov chains. *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING*, 29(7):2003, 2003.

[31] Thomas Ball, Byron Cook, Vladimir Levin, and SriramK. Rajamani. SLAM and Static Driver Verifier: Technology Transfer of Formal Methods inside Microsoft. In *Integrated Formal Methods*, volume 2999 of *Lecture Notes in Computer Science*, pages 1–20. Springer, 2004.

[32] Thomas Ball, Rupak Majumdar, Todd Millstein, and Sriram K. Rajamani. Automatic Predicate Abstraction of C Programs. *SIGPLAN Not.*, 36(5):203–213, May 2001.

[33] Richard Banach and Michael Butler. Cruise control in hybrid event-b. note: Unpublished, September 2013.

[34] Richard Banach, Huibiao Zhu, Wen Su, and Xiaofeng Wu. Continuous Behaviour in Event-B: A Sketch. In John Derrick, John Fitzgerald, Stefania Gnesi, Sarfraz Khurshid, Michael Leuschel, Steve Reeves, and Elvinia Riccobene, editors, *Abstract State Machines, Alloy, B, VDM, and Z*, volume 7316 of *Lecture Notes in Computer Science*, pages 349–352. Springer Berlin Heidelberg, 2012.

[35] S Serge Barold, Roland Stroobandt, and Alfons F Sinnaeve. *Cardiac Pacemakers and Resynchronization Step-by-Step: An Illustrated Guide*. Wiley-Blackwell, 2010.

[36] Bernhard Beckert, Thorsten Bormer, Florian Merz, and Carsten Sinz. Integration of Bounded Model Checking and Deductive Verification. In *Proceedings of the 2011 International Conference on Formal Verification of Object-Oriented Software*, FoVeOOS'11, pages 86–104, Turin, Italy, 2012. Springer-Verlag.

[37] Patrick Behm, Paul Benoit, Alain Faivre, and Jean-Marc Meynadier. Météor: A Successful Application of B in a Large Project. In Jeannette M. Wing, Jim Woodcock, and Jim Davies, editors, *FM'99 — Formal Methods: World Congress on Formal Methods in the Development of Computing Systems Toulouse, France, September 20–24, 1999 Proceedings, Volume I*, pages 369–387. Springer Berlin Heidelberg, Berlin, Heidelberg, 1999.

[38] Gerd Behrmann, Alexandre David, and Kim G. Larsen. A tutorial on uppaal. In Marco Bernardo and Flavio Corradini, editors, *Formal Methods for the Design of Real-Time Systems: 4th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM-RT 2004*, LNCS, pages 200–236. Springer–Verlag, September 2004.

[39] Eric Bodden, Patrick Lam, and Laurie Hendren. Clara: A Framework for Partially Evaluating Finite-State Runtime Monitors Ahead of Time. In Howard Barringer, Ylies Falcone, Bernd Finkbeiner, Klaus Havelund, Insup Lee, Gordon Pace, Grigore Roşu, Oleg Sokolsky, and Nikolai Tillmann, editors, *Runtime Verification*, volume 6418 of *Lecture Notes in Computer Science*, pages 183–197. Springer Berlin Heidelberg, 2010.

[40] F. Boniol, V. Wiels, Y.A. Ameur, and K.D. Schewe. *ABZ 2014: The Landing Gear Case Study: Case Study Track, Held at the 4th International Conference on Abstract State Machines, Alloy, B, TLA, VDM, and Z, Toulouse, France, June 2-6, 2014, Proceedings*. Communications in Computer and Information Science. Springer International Publishing, 2014.

[41] J.L. Boulanger. *Formal Methods Applied to Industrial Complex Systems: Implementation of the B Method*. ISTE. Wiley, 2014.

[42] Jonathan P. Bowen and Michael G. Hinchey. The Use of Industrial-strength Formal Methods. In *Proceedings of the 21st International Computer Software and Applications Conference*, COMPSAC '97, pages 332–337, Washington, DC, USA, 1997. IEEE Computer Society.

[43] J.W. Bryans, J.S. Fitzgerald, A. Romanovsky, and A. Roth. Patterns for Modelling Time and Consistency in Business Information Systems. In *Engineering of Complex Computer Systems (ICECCS)*, pages 105–114, Oxford, UK, March 2010. IEEE Computer Society.

[44] Didier Buchs, Steve Hostettler, Alexis Marechal, and Matteo Risoldi. AlPiNA: A Symbolic Model Checker. In Johan Lilius and Wojciech Penczek, editors, *Applications and Theory of Petri Nets*, volume 6128 of *Lecture Notes in Computer Science*, pages 287–296. Springer Berlin Heidelberg, 2010.

[45] Peter Bulychev, Alexandre David, Kim Gulstrand Larsen, Marius Mikučionis, Danny Bogsted Poulsen, Axel Legay, and Zheng Wang. UPPAAL-SMC: Statistical Model Checking for Priced Timed Automata. In Herbert Wiklicky and Mieke Massink, editors, *Proceedings 10th Workshop on Quantitative Aspects of Programming Languages and Systems, Tallinn, Estonia, 31 March and 1 April 2012*, volume 85 of *Electronic Proceedings in Theoretical Computer Science*, pages 1–16. Open Publishing Association, 2012.

[46] R. M. Burstall and John Darlington. Some Transformations for Developing Recursive Programs. *SIGPLAN Not.*, 10(6):465–472, April 1975.

[47] Rod M. Burstall. Program Proving as Hand Simulation with a Little Induction. In *IFIP Congress*, pages 308–312, 1974.

[48] Michael Butler. Decomposition Structures for Event-B. In *Integrated Formal Methods iFM2009, Springer, LNCS 5423*, volume LNCS. Springer, February 2009.

[49] Michael Butler and Jerome Falampin. An Approach to Modelling and Refining Timing Properties in B. In *Proceedings of Workshop on Refinement of Critical Systems (RCS)*, January 2002.

[50] Michael Butler and Issam Maamria. Practical Theory Extension in Event-B. In Zhiming Liu, Jim Woodcock, and Huibiao Zhu, editors, *Theories of Programming and Formal Methods*, volume 8051 of *Lecture Notes in Computer Science*, pages 67–81. Springer Berlin Heidelberg, 2013.

[51] R. W. Butler and G. B. Finelli. The infeasibility of quantifying the reliability of life-critical real-time software. *IEEE Transactions on Software Engineering*, 19(1):3–12, January 1993.

[52] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. EXE: Automatically generating inputs of death. In *In Proceedings of the 13th ACM Conference on Computer and Communications Security (CCS*, 2006.

[53] C. Callender. *The Oxford Handbook of Philosophy of Time.* Oxford Handbooks in Philosophy. OUP Oxford, 2011.

[54] Dominique Cansell and Dominique Méry. Foundations of the B method. *Computers and Informatics*, 22:31 p, 2003.

[55] Dominique Cansell, Dominique Méry, and Joris Rehm. Time Constraint Patterns for Event B Development. In *B 2007: Formal Specification and Development in B*, volume 4355 of *LNCS*, pages 140–154. Springer, 2006.

[56] Dubravka Cecez-Kecmanovic, Karlheinz Kautz, and Rebecca Abrahall. Reframing Success and Failure of Information Systems: A Performative Perspective. *MIS Q.*, 38(2):561–586, June 2014.

[57] K Mani Chandy. *Parallel Program Design: A Foundation.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1988.

[58] T. Chen, T. Han, and M. Kwiatkowska. On the complexity of model checking interval-valued discrete time Markov chains. *Inf. Process. Lett.*, 113(7):210–216, 2013.

[59] Shih-Tsung Cheng and Kuan-Hung Yeh. Pseudomalfunction of a dual chamber pacemaker caused by accelerated junctional rhythm and alternating ventricular safety pacing. *Europace*, 10(5):591–2, May 2008.

[60] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic Verification of Finite-state Concurrent Systems Using Temporal Logic Specifications. *ACM Trans. Program. Lang. Syst.*, 8(2):244–263, April 1986.

[61] Edmund M. Clarke and E. Allen Emerson. Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic. In *Logic of Programs, Workshop*, pages 52–71, London, UK, UK, 1982. Springer-Verlag.

[62] A Curtis. *Fundamentals of Cardiac Pacing.* Jones & Bartlett Learning, 2010.

[63] Barbara B Cuthill. Applicability of Object-Oriented Design Methods and C++ to Safety-critical Systems. In *Reliability and Nuclear Safety Workshop*, page 163, 1993.

[64] Dionisio De Niz. Diagrams and languages for model-based software engineering of embedded systems: UML and AADL. *White Paper, www. sei. cmu. edu/ library*, 2007.

[65] David Déharbe, Pascal Fontaine, Yoann Guyot, and Laurent Voisin. SMT Solvers for Rodin. In *Abstract State Machines, Alloy, B, VDM, and Z*, volume 7316 of *LNCS*, pages 194–207. Springer, 2012.

[66] E W Dijkstra. *A Discipline of Programming*. Prentice-Hall series in automatic computation. Prentice-Hall, Incorporated, 1976.

[67] Edsger W Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18(8):453–457, August 1975.

[68] Edsger W. Dijkstra, O. J. Dahl, E. W. Dijkstra, and C. A. R. Hoare. *Structured Programming*. Academic Press Ltd., London, UK, UK, 1972.

[69] A. Diller. *Z: An Introduction to Formal Methods*. Wiley & Sons, 1994.

[70] F. Neil Doherty, Colin Ashurst, and Joe Peppard. Factors affecting the successful realisation of benefits from systems development projects: Findings from three case studies. *Journal of Information Technology*, 27(1):1–16, 2012.

[71] Dominique Méry and Neeraj Kumar Singh. Pacemaker's Functional Behaviors in Event-B. Technical Report INRIA-00419973, INRIA Lorraine, LORIA - Laboratoire Lorrain de Recherche en Informatique et ses Applications, 2009.

[72] Mark Dowson. The Ariane 5 Software Failure. *SIGSOFT Softw. Eng. Notes*, 22(2):84–, March 1997.

[73] Andrew Edmunds and Michael Butler. Tasking Event-B: An Extension to Event-B for Generating Concurrent Code. Event Dates: 2nd April 2011, February 2011.

[74] Hilding Elmqvist, Sven Erik Mattsson, and Martin Otter. Modelica: The New Object-Oriented Modeling Language. In *12th European Simulation Multiconference, Manchester, UK*, 1998.

[75] E. A. Emerson, A. K. Mok, A. P. Sistla, and J. Srinivasan. Quantitative temporal reasoning. *Real-Time Systems*, 4(4):331–352, 1992.

[76] E. Allen Emerson. Temporal and modal logic. In *HANDBOOK OF THEORETICAL COMPUTER SCIENCE*, pages 995–1072. Elsevier, 1995.

[77] A. S. Evans, D. R. W. Holton, L. M. Lai, and P. Watson. A Comparison of Formal Real-time Specification Languages. In *Proceedings of the 1st BCS-FACS Conference on Northern Formal Methods*, 1FACS'96, pages 7–7, Ilkley, UK, 1996. BCS Learning & Development Ltd.

[78] Faezeh Siavashi, Marina Waldén, Leonidas Tsiopoulos, and Jüri Vain. Modelling Critical Systems with Time Constraints in Event-B. *Proceedings of the 25th Nordic Workshop on Programming Theory, NWPT '13*, 2013.

[79] Yliès Falcone, Klaus Havelung, and Giles Reger. A Tutorial on Runtime Verification. In Georg Kalus Manfred Broy, Doron Peled, editor, *Engineering Dependable Software Systems*, volume 34 of *NATO Science for Peace and Security Series - D: Information and Communication Security*, pages 141–175. IOS Press, 2013. Summer School Marktoberdorf 2012.

[80] John Fitzgerald. Approaches to the Pacemaker Challenge Problem. September 2008.

[81] John S. Fitzgerald, Peter Gorm Larsen, and Marcel Verhoef. Vienna Development Method. In *Wiley Encyclopedia of Computer Science and Engineering*. John Wiley Sons, Inc., 2007.

[82] Andreas Fürst, Thai Son Hoang, David Basin, Krishnaji Desai, Naoto Sato, and Kunihiko Miyazaki. Code Generation for Event-B. In Elvira Albert and Emil Sekerinski, editors, *Integrated Formal Methods: 11th International Conference, IFM 2014, Bertinoro, Italy, September 9-11, 2014, Proceedings*, pages 323–338. Springer International Publishing, Cham, 2014.

[83] M.C.W. Geilen. *Formal Techniques for Verification of Complex Real-Time Systems*. Technische Universiteit Eindhoven, 2002.

[84] Susan L Gerhart. Correctness-preserving program transformations. In *Proceedings of the 2nd ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 54–66. ACM, 1975.

[85] Carlo Ghezzi, Dino Mandrioli, Sandro Morasca, and Mauro Pezzè. A Unified High-Level Petri Net Formalism for Time-Critical Systems. *IEEE Trans. Softw. Eng.*, 17(2):160–172, February 1991.

[86] Antoine Girard and George J. Pappas. Verification Using Simulation. In João P. Hespanha and Ashish Tiwari, editors, *Hybrid Systems: Computation and Control: 9th International Workshop, HSCC 2006, Santa Barbara, CA, USA, March 29-31, 2006. Proceedings*, pages 272–286. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006.

[87] Martin Gogolla. Benefits and Problems of Formal Methods. In Albert Llamosí and Alfred Strohmeier, editors, *Reliable Software Technologies - Ada-Europe 2004*, volume 3063 of *Lecture Notes in Computer Science*, pages 1–15. Springer Berlin Heidelberg, 2004.

[88] Susanne Graf and Andreas Prinz. Time in State Machines. *Fundam. Inf.*, 77(1-2):143–174, January 2007.

[89] A. Hall and D. Isaac. Formal methods in a real air traffic control project. In *Software in Air Traffic Control Systems - The Future, IEE Colloquium On*, pages 7/1–7/4, June 1992.

[90] Anthony Hall. Realising the Benefits of Formal Methods. In Kung-Kiu Lau and Richard Banach, editors, *Formal Methods and Software Engineering*, volume 3785 of *Lecture Notes in Computer Science*, pages 1–4. Springer Berlin Heidelberg, 2005.

[91] Dominik Hansen, Lukas Ladenberger, Harald Wiegard, Jens Bendisposto, and Michael Leuschel. Validation of the ABZ Landing Gear System Using ProB. In Frédéric Boniol, Virginie Wiels, Yamine Ait Ameur, and Klaus-Dieter Schewe, editors, *ABZ 2014: The Landing Gear Case Study*, volume 433 of *Communications in Computer and Information Science*, pages 66–79. Springer International Publishing, 2014.

[92] D L Hayes, P A Friedman, and M Lloyd. *Cardiac Pacing and Defibrillation: A Clinical Approach*. John Wiley & Sons, 2000.

[93] C. Heitmeyer and N. Lynch. The generalized railroad crossing: A case study in formal verification of real-time systems. In *1994 Proceedings Real-Time Systems Symposium*, pages 120–131, December 1994.

[94] C. L. Heitmeyer, B. G. Labaw, and R. D. Jeffords. A benchmark for comparing different approaches for specifying and verifying real-time systems. In *IN PROC. 10 TH IEEE WORKSHOP ON REAL-TIME OPERATING SYSTEMS AND SOFTWARE*, 1993.

[95] Tom Henzinger, Zohar Manna, and Amir Pnueli. Temporal Proof Methodologies for Real-Time Systems. In *Proceedings of the 18th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 353–366. ACM, 1991.

[96] Timothy L. Hinrichs, A. Prasad Sistla, and Lenore D. Zuck. Model Check What You Can, Runtime Verify the Rest. In Andrei Voronkov and Margarita Korovina, editors, *HOWARD-60. A Festschrift on the Occasion of Howard Barringer's 60th Birthday*, pages 234–244. EasyChair, 2014.

[97] C A R Hoare. An Axiomatic Basis for Computer Programming. *Commun. ACM*, 12(10):576–580, October 1969.

[98] Gerard Holzmann. *Spin Model Checker, the: Primer and Reference Manual*. Addison-Wesley Professional, first edition, 2003.

[99] Gerard J. Holzmann. *The SPIN Model Checker - Primer and Reference Manual*. Addison-Wesley, 2004.

[100] Aram Hovsepyan, Dimitri Van Landuyt, Steven Op de beeck, Sam Michiels, Wouter Joosen, Gustavo Rangel, Javier Fernandez Briones, and Jan Depauw. Model-driven software development of safety-critical avionics systems: An experience report. In *1st International Workshop on Model-Driven Development Processes and Practices Co-Located with ACM/IEEE 17th International Conference on Model Driven Engineering Languages & Systems (MoDELS 2014)*, volume 1249, September 2014.

[101] HugoDaniel Macedo, PeterGorm Larsen, and John Fitzgerald. Incremental Development of a Distributed Real-Time Model of a Cardiac Pacing System Using VDM. In *FM 2008: Formal Methods*, volume 5014 of *LNCS*, pages 181–197. Springer, 2008.

[102] Michael Huth and Mark Ryan. *Logic in Computer Science: Modelling and Reasoning About Systems*. Cambridge University Press, New York, NY, USA, 2004.

[103] Dubravka Ilić, Elena Troubitsyna, Linas Laibinis, and Colin Snook. Formal Development of Mechanisms for Tolerating Transient Faults. In Michael Butler, CliffB. Jones, Alexander Romanovsky, and Elena Troubitsyna, editors, *Rigorous Development of Complex Fault-Tolerant Systems*, volume 4157 of *Lecture Notes in Computer Science*, pages 189–209. Springer Berlin Heidelberg, 2006.

[104] Jonathan Jacky. Specifying a Safety-Critical Control System in Z. *IEEE Trans. Softw. Eng.*, 21(2):99–106, February 1995.

[105] F Jahanian and A K Mok. Safety Analysis of Timing Properties in Real-time Systems. *IEEE Trans. Softw. Eng.*, 12(9):890–904, September 1986.

[106] Eunkyoung Jee, Shaohui Wang, Jeong Ki Kim, Jaewoo Lee, O. Sokolsky, and Insup Lee. A Safety-Assured Development Approach for Real-Time Software. In *The Proceedings of the 16th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 133–142, August 2010.

[107] Jesper Berthing, Pontus Boström, Kaisa Sere, Leonidas Tsiopoulos, and Jüri Vain. Refinement-Based Development of Timed Systems. In *Integrated Formal Methods*, volume 7321 of *LNCS*, pages 69–83. Springer, 2012.

[108] Capers Jones. Software project management practices: Failure versus success. *CrossTalk: The Journal of Defense Software Engineering*, 17(10):5–9, 2004.

[109] Joris Rehm. From Absolute-Timer to Relative-Countdown: Patterns for Model-Checking. Unpublished. note: Unpublished, May 2008.

[110] Kenneth Evensen and Kathryn Anne Weiss. A comparison and evaluation of real-time software systems modeling languages. In *JPL TRS 1992+*. Jet Propulsion Laboratory, National Aeronautics and Space Administration, 2010.

[111] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and WilliamG. Griswold. An Overview of AspectJ. In JørgenLindskov Knudsen, editor, *ECOOP 2001 — Object-Oriented Programming*, volume 2072 of *Lecture Notes in Computer Science*, pages 327–354. Springer Berlin Heidelberg, 2001.

[112] Gerwin Klein, June Andronick, Kevin Elphinstone, Gernot Heiser, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal Verification of an Operating-system Kernel. *Commun. ACM*, 53(6):107–115, June 2010.

[113] J.C. Knight, C.L. DeJong, M.S. Gibble, and L.G. Nakano. Why are formal methods not used more widely? In *Fourth NASA Formal Methods Workshop*. Citeseer, 1997.

[114] John C. Knight. Safety Critical Systems: Challenges and Directions. In *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02, pages 547–550, Orlando, Florida, 2002. ACM.

[115] Hermann Kopetz. *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Kluwer Academic Publishers, Norwell, MA, USA, 1st edition, 1997.

[116] D.G. Kourie and B.W. Watson. *The Correctness-by-Construction Approach to Programming*. Springer, 2012.

[117] Marta Kwiatkowska, Gethin Norman, and David Parker. Stochastic Model Checking. In *Proceedings of the 7th International Conference on Formal Methods for Performance Evaluation*, SFM'07, pages 220–270, Bertinoro, Italy, 2007. Springer-Verlag.

[118] Luming Lai and Phil Watson. A case study in timed CSP: The railroad crossing problem. In Oded Maler, editor, *Hybrid and Real-Time Systems: International Workshop, HART'97 Grenoble, France, March 26–28, 1997 Proceedings*, pages 69–74. Springer Berlin Heidelberg, Berlin, Heidelberg, 1997.

[119] Monica S. Lam, Michael Martin, Benjamin Livshits, and John Whaley. Securing Web Applications with Static and Dynamic Information Flow Tracking. In *Proceedings of the 2008 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, PEPM '08, pages 3–12, San Francisco, California, USA, 2008. ACM.

[120] Leslie Lamport. Introduction to TLA. *Digital Equipment Corporation Systems Research Center [SRC]*, 1994.

[121] Leslie Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[122] Kim G Larsen, Paul Pettersson, and Wang Yi. Uppaal in a Nutshell. *International Journal on Software Tools for Technology Transfer (STTT)*, 1(1):134–152, 1997.

[123] Gary T Leavens, Albert L Baker, and Clyde Ruby. JML: A notation for detailed design. In *Behavioral Specifications of Businesses and Systems*, pages 175–188. Springer, 1999.

[124] Edward A. Lee and Sanjit A. Seshia. *Introduction to Embedded Systems - A Cyber-Physical Systems Approach*. Lee and Seshia, 1 edition, 2010.

[125] Michael Leuschel and Michael Butler. ProB: A Model Checker for B. In *FME 2003: Formal Methods*, volume 2805 of *LNCS*, pages 855–874. Springer, 2003.

[126] N. G. Leveson and C. S. Turner. An Investigation of the Therac-25 Accidents. *Computer*, 26(7):18–41, July 1993.

[127] Magnus Lindahl, Paul Pettersson, and Wang Yi. Formal Design and Analysis of a Gear Controller. *International Journal on Software Tools for Technology Transfer*, 3(3):353–368, 2001.

[128] M. R. Sena Marques, E. Siegert, and L. Brisolara. Integrating UML, MARTE and sysml to improve requirements specification and traceability in the embedded domain. In *2014 12th IEEE International Conference on Industrial Informatics (INDIN)*, pages 176–181, July 2014.

[129] Stephen J. Mellor and Marc Balcer. *Executable UML: A Foundation for Model-Driven Architectures*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[130] Dominique Méry and Neeraj Kumar Singh. Automatic Code Generation from event-B Models. In *Proceedings of the Second Symposium on Information and Communication Technology*, SoICT '11, pages 179–188, Hanoi, Vietnam, 2011. ACM.

[131] Bertrand Meyer. Applying "Design by Contract". *Computer*, 25(10):40–51, October 1992.

[132] Marius Mikcionis, Kim Guldstrand Larsen, Jacob Illum Rasmussen, Brian Nielsen, Arne Skou, Steen Ulrik Palm, Jan Storbank Pedersen, and Poul Hougaard. Schedulability Analysis Using Uppaal: Herschel-Planck Case Study. In *Proceedings of the*

4th International Conference on Leveraging Applications of Formal Methods, Verification, and Validation - Volume Part II, ISoLA'10, pages 175–190, Heraklion, Crete, Greece, 2010. Springer-Verlag.

[133] Carrol Morgan, Ken Robinson, and Paul Gardiner. *On the Refinement Calculus.* Oxford University. Computing Laboratory. Programming Research Group, 1988.

[134] Glenford J. Myers and Corey Sandler. *The Art of Software Testing.* John Wiley & Sons, 2004.

[135] Chris Newcombe. Why Amazon Chose TLA. In Yamine Ait Ameur and Klaus-Dieter Schewe, editors, *Abstract State Machines, Alloy, B, TLA, VDM, and Z: 4th International Conference, ABZ 2014, Toulouse, France, June 2-6, 2014. Proceedings*, pages 25–39. Springer Berlin Heidelberg, Berlin, Heidelberg, 2014.

[136] Jonathan S. Ostroff. *Temporal Logic for Real Time Systems.* John Wiley & Sons, Inc., New York, NY, USA, 1989.

[137] Paul Pettersson. Formal Methods Applied in Industry - On the Commercialisation of the UPPAAL Tool. In *COMPSAC'11*, pages 450–451, 2011.

[138] Amir Pnueli. The Temporal Semantics of Concurrent Programs. In *Proceedings of the International Sympoisum on Semantics of Concurrent Computation*, pages 1–20, London, UK, UK, 1979. Springer-Verlag.

[139] J.N. Reed, J.E. Sinclair, and F. Guigand. Deductive Reasoning versus Model Checking: Two Formal Approaches for System Development. In Keijiro Araki, Andy Galloway, and Kenji Taguchi, editors, *IFM'99*, pages 375–394. Springer London, 1999.

[140] Víctor Rivera. Code Generation for Event-B. *CoRR*, abs/1602.02004, 2016.

[141] John Rushby. Formal Methods and Critical Systems in the Real World. In Dan Craigen and Karen Summerskill, editors, *Formal Methods for Trustworthy Computer Systems (FM89)*, pages 121–125, Halifax, Nova Scotia, Canada, July 1989. Springer-Verlag Workshops in Computing.

[142] Yavin Sa'ar. *Deductive and Algorithmic Methods for Formal Verification.* PhD thesis, Weizmann Institute of Science, 2007.

[143] Mohammad Reza Sarshogh. *Extending Event-B with Discrete Timing Properties.* PhD thesis, University of Southampton, 2013.

[144] Steve Schneider. *Concurrent and Real Time Systems: The CSP Approach.* John Wiley & Sons, Inc., New York, NY, USA, 1st edition, 1999.

[145] Sally Shlaer. The shlaer-mellor method. *Project Technology white paper*, 1996.

[146] Sally Shlaer and Stephen J. Mellor. *Object-Oriented Systems Analysis: Modeling the World in Data*. Yourdon Press, Upper Saddle River, NJ, USA, 1988.

[147] J. A. Stankovic. Misconceptions about real-time computing: A serious problem for next-generation systems. *Computer*, 21(10):10–19, October 1988.

[148] Gintautas Sulskus, Michael Poppleton, and Abdolbaghi Rezazadeh. An Interval-Based Approach to Modelling Time in Event-B. In *Fundamentals of Software Engineering*, volume 9392 of *Lecture Notes in Computer Science*, pages 292–307. Springer, 2015.

[149] Gintautas Sulskus, Michael Poppleton, and Abdolbaghi Rezazadeh. Modelling Complex Timing Requirements with Refinement. In *Information Reuse and Integration (IRI), 2016 IEEE 17th International Conference On*, July 2016.

[150] R.J.M.D. Sung and M.R.M.D. Lauer. *Fundamental Approaches to the Management of Cardiac Arrhythmias*. Kluwer Academic Pub, 2000.

[151] N. Tanković, D. Vukotić, and M. Žagar. Rethinking Model Driven Development: Analysis and opportunities. In *Information Technology Interfaces (ITI), Proceedings of the ITI 2012 34th International Conference On*, pages 505–510, June 2012.

[152] Jonathan Timperley, Paul Leeson, Andrew Rj Mitchell, and Timothy Betts. *Pacemakers and ICDs*. Oxford University Press, 2007.

[153] GAO United States General Accounting Office. Patriot Missile Defence. Software Problem Led to System Failure at Dhahran, Saudi Arabia. http://www.gao.gov/products/IMTEC-92-26, 1992.

[154] Eric Verhulst and Gjalt de Jong. OpenComRTOS: An Ultra-Small Network Centric Embedded RTOS Designed Using Formal Modeling. In Emmanuel Gaudin, Elie Najm, and Rick Reed, editors, *SDL 2007: Design for Dependable Systems: 13th International SDL Forum Paris, France, September 18-21, 2007 Proceedings*, pages 258–271. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007.

[155] Kostyantyn Vorobyov and Padmanabhan Krishnan. Comparing Model Checking and Static Program Analysis: A Case Study in Error Detection Approaches. *Systems Software Verification*, 2010.

[156] Jiacun Wang. *Handbook of Finite State Based Models and Applications*. Discrete Mathematics and Its Applications. Chapman And Hall/CRC, 2012.

[157] R. J. Wieringa and G. Saake. Formal analysis of the Shlaer-Mellor method: Towards a toolkit of formal and informal requirements specification techniques. *Requirements Engineering*, 1(2):106–131, 1996.

[158] Reinhard Wilhelm, Sebastian Altmeyer, Claire Burguière, Daniel Grund, Jörg Herter, Jan Reineke, Björn Wachter, and Stephan Wilhelm. Static Timing Analysis for Hard Real-Time Systems. In Gilles Barthe and Manuel Hermenegildo, editors, *Verification, Model Checking, and Abstract Interpretation*, volume 5944 of *Lecture Notes in Computer Science*, pages 3–22. Springer Berlin Heidelberg, 2010.

[159] Niklaus Wirth. Program development by stepwise refinement. *Communications of the ACM*, 14(4):221–227, 1971.

[160] Jim Woodcock and Jim Davies. *Using Z: Specification, Refinement, and Proof.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996.

[161] Jim Woodcock, Peter Gorm Larsen, Juan Bicarregui, and John Fitzgerald. Formal Methods: Practice and Experience. *ACM Comput. Surv.*, 41(4):19:1–19:36, October 2009.

[162] S. Yamane. Deductive verification of probabilistic real-time systems. In *Distributed Computing Systems Workshops, 2004. Proceedings. 24th International Conference On*, pages 622–627, March 2004.

[163] Faqing Yang and Jean-Pierre Jacquot. Scaling Up with Event-B: A Case Study. In *NASA Formal Methods*, volume 6617 of *LNCS*, pages 438–452. Springer, 2011.

[164] Zhihao Jiang, Miroslav Pajic, Salar Moarref, Rajeev Alur, and Rahul Mangharam. Modeling and Verification of a Dual Chamber Implantable Pacemaker. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 7214 of *LNCS*, pages 188–203. Springer, 2012.