UNIVERSITY OF SOUTHAMPTON

Formal Treatment of Real-time Properties in Event-B

by

Chenyang Zhu

Thesis for the degree of Doctor of Philosophy

in the
Faculty of Engineering and Physical Sciences
School of Electronics and Computer Science

July 2020

UNIVERSITY OF SOUTHAMPTON

ABSTRACT

Faculty of Engineering and Physical Sciences
School of Electronics and Computer Science

Doctor of Philosophy

by Chenyang Zhu

Timing and concurrency are two critical properties of Cyber-Physical Systems (CPS). Functional and timing requirements needed to be satisfied in CPS to avoid unsafe situations. Formal methods, which are mathematical techniques for specifying and verifying systems, aid software engineering by ensuring the correctness of the system design.

The Event-B formalism offers a stepwise development approach to manage complexity in system design. Our work provides formal treatment of real-time properties in Event-B models from both the semantics perspective and syntax perspective. There is existing work on treating real-time properties in Event-B but it lacks a semantic treatment in terms of trace behaviors. Because timing properties require fairness assumptions, we use infinite traces and develop conditions under which all infinite traces of a machine satisfy trigger-response and timing properties. We present refinement semantics of models whose behavior traces are infinite. Based on forward simulation, fairness assumptions, relative deadlock freedom, and conditional convergence are adopted as additional conditions that guarantee infinite trace refinement of timed models.

Also, the existing work that extends Event-B models with discrete timing properties inadequately represents the communication and competition between concurrent tasks in concurrent systems. We present the semantics of parameterized real-time trigger-response properties of Event-B models based on timing invariants. We show a method of syntactically encoding parameterized real-time trigger-response properties in Event-B machines. To capture the concurrency between tasks, we distinguish end-to-end timing properties and scheduler-based timing properties from the perspective of different system design phases. We model end-to-end timing properties as parameterized timing properties and scheduler-based timing properties as unparameterized timing properties. A nondeterministic queue-based scheduling framework is proposed to replace end-to-end timing properties with scheduler-based timing properties.

Finally, we demonstrate our approach with three real-time case studies. We show how to treat real-time properties in a stepwise modeling and verification process with Event-B models.

Contents

List of Figures

List of Tables

List of Symbols

$S \times T$	Cartesian product	$S \times T = \{x \mapsto y \mid x \in S \land y \in T\}$
$\mathbb{P}(S)$	Powerset	$\mathbb{P}(S) = \{s \mid s \subseteq S\}$
$S \leftrightarrow T$	Relations	$S \leftrightarrow T = \mathbb{P}(S \times T)$
$m..n$	Interval	$m..n = \{i \mid m \leq i \land i \leq n\}$
$\mathrm{dom}(r)$	Domain	$\forall r \cdot r \in S \leftrightarrow T \Rightarrow \mathrm{dom}(r) = \{x \cdot (\exists y \cdot x \mapsto y \in r)\}$
$\mathrm{ran}(r)$	Range	$\forall r \cdot r \in S \leftrightarrow T \Rightarrow \mathrm{ran}(r) = \{y \cdot (\exists x \cdot x \mapsto y \in r)\}$
$p; q$	Forward composition	$\forall p, q \cdot p \in S \leftrightarrow T \land q \in T \leftrightarrow U \Rightarrow$ $p; q = \{x \mapsto y \mid (\exists z \cdot x \mapsto z \in p \land z \mapsto y \in q)\}$
$p \circ q$	Backward composition	$p \circ q = q; p$
id	Identity	$S \lhd id = \{x \mapsto x \mid x \in S\}$
$r \unrhd T$	Range subtraction	$r \unrhd T = \{x \mapsto y \mid y \in r \land y \notin T\}$
$S \lhd r$	Domain restriction	$S \lhd r = \{x \mapsto y \mid x \mapsto y \in r \land x \in S\}$
$S \unlhd r$	Domain subtraction	$S \unlhd r = \{x \mapsto y \mid x \mapsto y \in r \land x \notin S\}$
r^{-1}	Inverse	$r^{-1} = \{y \mapsto x \mid x \mapsto y \in r\}$
$r[S]$	Relational image	$r[S] = \{y \mid \exists x \cdot x \in S \land x \mapsto y \in r\}$
$r_1 \Leftarrow r_2$	Overriding	$r_1 \Leftarrow r_2 = r_2 \cup (\mathrm{dom}(r_2) \unlhd r_1)$
$S \nrightarrow T$	Partial functions	$S \nrightarrow T = \{r \cdot r \in S \leftrightarrow T \land r^{-1}; r \subseteq T \lhd id\}$
$S \rightarrow T$	Total functions	$S \rightarrow T = \{f \cdot f \in S \nrightarrow T \land \mathrm{dom}(f) = S\}$
$S \rightarrowtail T$	Total injections	$S \rightarrowtail T = \{f \cdot f \in S \nrightarrow T \land f^{-1} \in T \nrightarrow S\} \cap S \rightarrow T$
$S \twoheadrightarrow T$	Total surjections	$S \twoheadrightarrow T = \{f \cdot f \in S \nrightarrow T \land \mathrm{ran}(f) = T\} \cap S \rightarrow T$
$S \rightarrowtail\!\!\!\!\twoheadrightarrow T$	Bijections	$S \rightarrowtail\!\!\!\!\twoheadrightarrow T = S \rightarrowtail T \cap S \twoheadrightarrow T$

Declaration of Authorship

I, Chenyang Zhu , declare that the thesis entitled 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: Zhu et al. (2018b), Zhu et al. (2018a), Zhu et al. (2019b), Zhu et al. (2020a), Zhu et al. (2020b), Zhu et al. (2020c)

Signed:..

Date:...

Acknowledgements

Firstly, I would like to express my gratitude to my supervisors Professor Michael Butler and Dr. Corina Cirstea, for their continuous support and guidance for four years. They have always been inspiring and uplifting and guide me to find the right solutions to different tackles. I appreciate all the efforts they made to help me walk through my Ph.D. life.

I would like to extend my thanks the researchers in the Cyber-Physical System group, Dr. Thai Son Hoang, Dr. Abdolbaghi Rezazadeh, Dr. Dana Dghaym, Dr. Colin Snook, Dr. Asieh Salehi Fathabadi. I would like to thank Dr. Ping Hua in ORC group. It is nice to work with them and enjoy my Ph.D. life.

I want to thank my friends for their company during my study, Yang Shi, Chenyan Xu, Jie Zhan, Runshan Hu, Jia Bi, Qian Ding.

Most importantly, I would like to give a special thank you to my parents Zhengwei Zhu, Junmin Zhi, and my wife, Yunxin Xie. They have always been supportive and encouraging, without whom I would never achieve this much. Especially thank my wife for bringing our first baby Ruixi to the world, which is the best gift for my life. Thanks for caring our baby girl alone when I'm working on the papers and research.

This thesis is supported in part by the scholarship from China Scholarship Council (CSC) under the Grant CSC NO.201708060147 and studentship of Department of Electronics and Computer Science of the University of Southampton under project 5160091087294.

Chapter 1

Introduction

Cyber-Physical Systems (CPS), which incorporate advanced processors, sensors and wireless communication technology, have received a significant amount of attention in recent years owing to their ability to interact with the physical world. CPS is a generalization that refers to embedded systems that consist of a collection of computing devices communicating with one another and interacting with the physical world via sensors and actuators in a feedback loop (Alur, 2015). CPS can be used in a wide range of fields such as medical monitoring, intelligent transportation and distributed robotics (Shi et al., 2011; Ahmed et al., 2013). Some of these applications, such as medical devices and aircraft flight control, are safety-critical, meaning that CPS failure could result in loss of life or property damage (Knight, 2002).

Timing and concurrency are two critical factors in the implementation of CPS. Timing properties should be specified when developing the system to guarantee that CPS is interacting with the environment correctly. Some safety-critical systems must satisfy certain real-time constraints in order to function as intended. For example, an artificial pacemaker should deliver therapy according to the misbehavior of the heart at fixed time intervals. Missing the stimulus deadlines could be catastrophic. Similarly, the flight control software must be able to distinguish between different flying environments and respond to different situations within a specific timeframe in order to control the aircraft safely. Thus CPS must fulfill its functional role in a time-regulated manner in order to avoid unsafe situations. In addition, the distributed computing devices in CPS function concurrently in order to work as a whole and achieve the computation goal of the whole system. The design and analysis of CPS are more challenging regarding the concurrent functioning of real-time applications when the system execute a set of concurrent tasks so that all time-critical tasks meet specific deadlines (Choudary et al., 1996). In concurrent systems, the Worst-Case Execution Time (WCET) is determined not only by the task itself but also the behavior of other tasks that share the same resources. Delays caused by the task preemption could also affect the whole performance of the system (Kopetz, 1997). In CPS, real-time scheduling is often used to optimize the real-time performance of the whole system by assigning executable orders to guarantee that all tasks meet their deadlines (Kim et al., 2012).

With the thriving growth of software technology, much attention has been concentrated on simplifying the design of software systems while maintaining their usability and dependability. However, the number of system errors increases in proportion to the scale and complexity of the whole system. Formal methods, which include mathematical techniques for specifying and verifying systems, help the developers to construct reliable software systems despite the design complexity (Clarke and Wing, 1996). Woodcock et al. (2009a) concluded that despite the considerable time and cost needed to develop industry systems using formal methods, 92% of the industry cases in the research report an increase in quality compared to other techniques. Formal methods have been used in different software development life-cycles: requirements specification, system design, and verification (Woodcock et al., 2009b). Tassey (2002) showed that system errors detected during integration and system test stages cost ten times more to fix compared with those detected in the requirements specification stage. Thus the early detection of errors or bugs could help to reduce the cost of software development. However, it is often challenging to get the specification of the system requirements right, and design errors are hard to identify during the early stages of the development of a software-based system (Butler, 2017). Formal modeling that provides mathematical abstractions can be used to manage the complexity of the system design.

In addition to precise mathematical descriptions, formal modeling is also supported by verification methods and tools to eliminate design errors in models. However, precise properties alone do not fully address the problem of modeling large and complicated systems such as CPS. It is difficult to model a complicated system that incorporates a range of detailed features in one step using mathematical abstraction. Instead, a stepwise approach can be adopted to develop formal models to manage the complexity of system design. In the abstract model, an abstraction of the system that focuses on its intended purpose can be used to simplify the understanding of the system (Butler, 2013a). Further stepwise refinement models can be used to determine how the intended purpose is achieved. Formal proofs and invariants can then be used to verify the consistency between different refinement levels.

Woodcock et al. (2009a) also demonstrated that the demand for modeling real-time applications using formal techniques is substantial. However, the fundamental problem for modeling CPS is that it relies on bridging the gap between the modeler and the domain expert, who is able to specify the functional requirements and the time constraints. Formal specifications allow the system designer to specify the model of real-time systems and make timing assertions relating to the system while leaving the verification and validation to the formal techniques and tools (Choudary et al., 1996). Work has been carried out to incorporate formal methods with real-time properties, including temporal logic (Ostroff, 1989; Alur and Henzinger, 1990), timed automata (Alur and

Dill, 1994), timed Petri-Nets (Coolahan and Roussopoulos, 1983; Gehlot, 1988), timed transition systems (Henzinger et al., 1991) and state-based approaches such as B (Butler and Falampin, 2002) method and Event-B (Sarshogh and Butler, 2011; Sarshogh, 2013; Sulskus et al., 2016; Sulskus, 2017).

1.1 Motivation

Safety and liveness are two key properties of formal models: a safety property states that dangerous situations will not arise while a liveness property ensures that something good will eventually happen (Lamport, 1977). Alpern and Schneider (1985) showed that all properties exist at the intersection of safety and liveness properties. In real-time systems, time should progress regardless of what happens in its environment (Ostroff, 1999). Thus liveness properties should be reasoned together with the real-time systems. Also, we are reasoning that the behavior traces are infinite since time should always progress. However, the existing refinement framework in Event-B models does not provide sufficient conditions and assumptions to confirm the trace inclusion of infinite behavioral traces. Regarding infinite behavioral traces, each refinement step requires that the behavior of a refined model should be consistent with the behavior of the model being refined. However, Event-B models are not concerned with fairness or scheduling specifications (Méry and Poppleton, 2015). The divergence of introduced new events or a deadlock in the concrete model would fail the infinite trace simulation.

Work has been done to extend Event-B models with time constraints such as delay, deadline, expiry, and intervals (Sarshogh and Butler, 2011; Sarshogh, 2013; Sulskus et al., 2016; Sulskus, 2017). However the developments have failed to incorporate a proper treatment of critical issues in timed systems, namely, the divergence of intermediate events and infeasible responses caused by a lack of progress or conflicting timing constraints. In addition, there is a gap between the trace semantics of Event-B models and syntax that extends Event-B models with time constraints. Work needs to be done to bridge the gap with formal proofs.

Moreover, existing work does not distinguish between the timing properties of different system design phases and cannot show the communication and competition between concurrent tasks in concurrent or distributed systems. In real-time systems, there are always several tasks running concurrently. High-level time constraints for each task cannot guarantee the timing behavior of the whole system. Fairness and scheduling needed to be introduced to the model to guarantee that the time constraints of each task are satisfied.

To summarize, we mainly address the following research questions in this thesis:

- What are the sufficient conditions needed on top of existing refinement framework in Event-B models to conform the trace inclusion of infinite behavioral traces.
- What are the conditions and fairness assumptions to exclude *Zeno* behavior in timed systems.
- How to bridge the gap between real-time specifications with formal models.
- How to refine timing properties from different design phases and capture the communication and competition between concurrent tasks in concurrent or distributed systems.

1.2 Contributions

From the semantics perspective, we develop proof obligations on Event-B machines to address a vital issue in timed systems, namely divergence of intermediate events and infeasible responses caused by conflicting time constraints. Lynch and Vaandrager (1995) assumed a functional relation between concrete states and abstract states when reasoning about the consistency between different refinement levels in terms of infinite traces by using forward simulation. However, the functional relation assumption does not address all the real-world refinement cases as the concrete states might correspond to one or more abstract states. Thus our work generalizes the result by using a relational mapping between concrete states and abstract states to simulate infinite-state traces. We present infinite trace simulation with gluing relations between abstract and concrete state traces with formal proofs. In this thesis, we explore sufficient conditions under which the refinement is valid regarding infinite behavior traces. Sufficient conditions are also explored under which all the traces of an Event-B model satisfy the real-time trigger-response property, in the form of Event-B proof obligations and fairness assumptions. Based on the monotonicity of timing properties in Event-B models, we design a two-step refinement strategy to extend Event-B models with timing properties. The first step performs data refinement with no changes to the timing properties. The second step refines timing properties into sequential or alternative timing properties. A Bounded Retransmission Protocol (BRP) case study is used to present the two-step refinement strategy and refinement patterns for real-time systems with unparameterized timing properties.

From the real-time specification perspective, we present four real-time specification patterns, namely time response pattern, abort pattern, intermediate pattern, and periodic pattern, to support quantitative reasoning about time. Refinement patterns are also provided to refine abstract timing properties to concrete timing properties. The real-time specification patterns are used to model the timing cycles of a dual-chamber pacemaker.

Moreover, we distinguish end-to-end timing properties and scheduler-based timing properties from different system design phases. End-to-end timing properties are defined as high-level timing properties from the system requirement specification phase, which place discrete-time properties on individual tasks. To model the behavior of these concurrent tasks, we defined scheduler-based timing properties as concrete timing properties for the system design phase, which place discrete-time constraints on the scheduler that schedules the concurrent tasks. We propose a non-deterministic queue-based scheduling framework to model the behavior of schedulers. Tasks are placed in a non-deterministic position in the queue, and once a task enters the queue, it cannot be postponed forever. Additional gluing invariants based on the refinement rule are provided to use the framework to refine end-to-end deadline constraints with scheduler-based deadline constraints. A two-level hierarchical scheduling system is formalized to show the refinement of end-to-end timing properties to scheduler-based timing properties with the non-deterministic scheduling framework. The hierarchical scheduling system uses Time Division Multiplexing (TDM) as the global scheduler. Our work shows that local schedulers are compatible with different scheduling policies refined from the same scheduling framework.

1.3 Thesis Outline

This thesis is structured into the following chapters:

Chapter 2 overviews several formal modeling approaches, including formal specification languages, analysis tools, and refinement approaches. This chapter also reviews time and fairness modeling with different formal specification languages. We present more detail on the modeling and refinement approaches that extend Event-B models with timing properties. We then discuss the deficiency of the existing time modeling approach.

Chapter 3 introduces trace semantics of Event-B models. We provide sufficient conditions under which all traces in Event-B models satisfy the trigger-response property. Also, it provides a formal definition of generic refinement semantics regarding infinite behavior traces.

Chapter 4 provides refinement semantics for real-time properties. Proofs are also provided to refine abstract timing properties to sequential or alternative sub-timing properties.

Chapter 5 uses the Bounded Re-transmission Protocol to demonstrate the required conditions for trigger-response property, which also utilizes the two-step refinement strategy to model the real-time system.

Chapter 6 provides four real-time specification patterns to model time in real-world systems. Refinement patterns are also provided to refine abstract timing properties to concrete timing properties.

Chapter 7 models the timing cycles of a dual-chamber pacemaker to illustrate the usage of real-time specification patterns.

Chapter 8 presents the semantics of parameterized trigger-response properties with refinement rules that replace parameterized timing properties to unparameterized timing properties.

Chapter 9 distinguishes the timing properties from different design phases in a hierarchical real-time system. A non-deterministic queue-based scheduling framework is used based on the refinement rule to refine end-to-end timing properties to scheduler-based timing properties.

Chapter 10 summarizes our conclusions and outlines future work.

1.4 List of Publications

The contributions of the work in thesis have been published or accepted in the following papers:

1.4.1 Journal Papers

- Chenyang Zhu, Michael Butler, and Corina Cirstea. Formalizing hierarchical scheduling for refinement of real-time systems. Science of Computer Programming, page 102390, 2020. (Published)
- Chenyang Zhu, Michael Butler, and Corina Cirstea. Trace Semantics and Refinement Patterns for Real-time Properties in Event-B Models. Science of Computer Programming, page 102513, 2020. (Published)

1.4.2 Conference Papers

- Chenyang Zhu, Michael Butler, and Corina Cirstea. Refinement of timing constraints for concurrent tasks with scheduling. In Abstract State Machines, Alloy, B, TLA, VDM, and Z: ABZ 2018, volume 10817, pages 219–233. Springer, May 2018. (Published)

- Chenyang Zhu, Michael Butler, and Corina Cirstea. Semantics of real-time trigger-response properties in Event-B. In 2018 International Symposium on Theoretical Aspects of Software Engineering, Theoretical Aspects of Software Engineering 2018, Guangzhou, China, August 29-31, 2018, pages 150–155, 2018. (Published)
- Chenyang Zhu, Michael Butler, and Corina Cirstea. Towards refinement semantics of real-time trigger-response properties in Event-B. In 13th International Symposium on Theoretical Aspects of Software Engineering, Guilin, China, July 2019. (Published)
- Chenyang Zhu, Michael Butler, and Corina Cirstea. Real-time trigger-response properties for Event-B applied to the pacemaker. In 14th International Symposium on Theoretical Aspects of Software Engineering, July 2020. (Accepted)

Chapter 2

Timing And Fairness Modeling with Formal Methods

2.1 Introduction

Cyber-Physical Systems (CPS) are embedded systems that integrate computation and physical components seamlessly, with the aim of providing dependable, high-confidence services to human beings (Nsf, 2019). Since CPS are composed of both cyber components and physical components, software failures, sensors failures, or communication failures exist, some of which can lead to catastrophic consequences (Johnson and Mitra, 2009). Faults in the Ariane 5 application and its environmental requirements, for example, entailed a loss of 1.9 billion French Francs and a one-year delay for the Ariane 5 programme (Le Lann, 1997). Also, the software-related failure of some medical devices may lead to death or injury (Wallace and Kuhn, 2001). Recently, formal specification and verification have been gaining popularity among industrial-sized case studies into improving the dependability and availability of the increasingly sophisticated software (Gleirscher et al., 2019). In formal methods, formal specifications use mathematically defined syntax to define system properties such as functional behavior and timing behavior (Clarke and Wing, 1996). Formal verification is used to verify the correctness of the formal semantics that describes the behavior of the system. The formal proof needs to be sound with respect to the semantics to ensure that the system behavior satisfies the specification (Ostroff, 1992). Abstraction and refinement are used in formal methods to manage the complexity of the requirements (Butler, 2013b). Refinements can be used to replace the abstract data structures with concrete ones that are more easily implemented (Liskov et al., 1986) and add complexity to the system models in ways that consistency can be verified (Abrial, 2009). In this chapter, we highlight the Event-B formal specification language and two formal verification approaches, namely model checking and theorem proving. Several refinement techniques are also presented in this chapter.

2.2 Formal Verification

2.2.1 Model Checking

Model checking is one of the well-established approaches for formal verification. Given a finite-state model of a system, model checking performs an exhaustive state space search to check whether the model meets some desired property (Clarke and Wing, 1996). Model checking is guaranteed to terminate as the state space is finite. However, the system state space grows exponentially with the increase in the number of state variables in the system (Clarke et al., 2011).

Temporal logic is a formalism introduced by Pnueli to express formal requirements such as safety requirements and liveness requirements of reactive systems (Pnueli, 1977). Linear Temporal Logic (LTL) is one of the most used temporal logics that model time as a sequence of states (Pnueli, 1977). LTL uses four temporal operators, namely, $\Box$, $\Diamond$, $\mathcal{U}$ and $\bigcirc$. Given a property φ in the state trace s, $s \models \Box\,\varphi$ denotes that every state in the trace satisfies property φ; $\Diamond\,\varphi$ is satisfied when some state in the trace satisfies φ; $\bigcirc\,\varphi$ states that the next state in the trace satisfies φ. An LTL-formula is built up from three parts: a finite set of typed variables, basic logical operators and basic temporal operators. For example, $\Box\Diamond\,\varphi$ denotes that in a trace φ is satisfied repeatedly. And $\Diamond\Box\,\varphi$ denotes that in a trace φ will eventually always be satisfied. Together with these four temporal operators used by LTL, Computation Tree Logic (CTL) use two path quantifiers, $\forall$ and $\exists$, to express the system properties (Clarke et al., 1994). Given the property φ, $\forall\varphi$ denotes that φ holds for all computation paths and $\exists\varphi$ holds for some computation path. Model checking is supported by a number tools, such as SPIN (Holzmann Gerard, 2003), UPPAAL (Behrmann et al., 2011), PRISM (Hinton et al., 2006).

2.2.2 Theorem Proving

Theorem proving is another verification technique that uses mathematical formulas to express both the system and its desired properties (Clarke and Wing, 1996). The process of theorem proving is finding the a proof of the system properties based on the defined axioms and inference rules. Based on induction and deduction rules, theorem proving is independent of the size of the state space. The development of computer techniques enables automated theorem proving by combining the inference rules to get the correct proof (Bibel, 1987). However, proving a mathematical theorem may require deep understanding of the system properties as well as the inference rules. Several interactive theorem provers that involve both human and computers to generate proofs for specific properties, such as Isabelle (Paulson, 1994), Rodin (Abrial et al., 2010), CVC3 (Barrett and Tinelli, 2007) and Z3 theorem prover (De Moura and Bjorner, 2008).

2.3 Refinement

2.3.1 Refinement Mappings

In formal methods, transition systems are used to describe the behavior of discrete systems (Keller, 1976; Pnueli, 1977). A transition system is defined in Definition 2.1. The transition system uses a set of states S to interpret the set of variables in the system. The set of transitions L describes the behavior of the discrete system.

Definition 2.1 (Transition Systems (Abadi and Lamport, 1991))**.** A transition system T is a four-tuple $< V, S, I, K >$ consisting of

- V: a finite set V of variables;
- S: a set S of states, where each $s \in S$ is an interpretation of V;
- I: a subset $I \subseteq S$ of initial states;
- L: a finite set of transitions $L \in (S \leftrightarrow S)$;

The refinement calculus is a way of transforming program structure so that correctness is preserved. This is different to refinement mappings which are for transforming program state rather than structure. In software engineering, programs or systems can be constructed with stepwise refinement, where each refinement step preserves the correctness of the previous properties of the system. Refinement mappings and forward simulation have been proved to be useful in proving refinements (Lynch and Vaandrager, 1995). The refinement mapping of transition systems is defined in Definition 2.2. Similar notions of refinement mapping have been defined in (Lynch and Vaandrager, 1995; Abadi and Lamport, 1991). The function f is a valid refinement mapping provided the state traces of T_1 and T_2 can be mapped with the mapping function. The conditions provided in Definition 2.2 can be used to infer state trace equivalence between T_1 and T_2. Abadi and Lamport (1991) concluded that if there exists a refinement mapping from T_1 to T_2, then T_2 implements T_1.

Definition 2.2 (Refinement Mapping)**.** Given transition systems $T_1 =< V_1, S_1, I_1, K_1 >$ and $T_2 =< V_2, S_2, I_2, K_2 >$, a refinement mapping from T_1 to T_2 is a function f from S_1 to S_2 that satisfies:

- $\forall s \cdot f(s) \in I_2 \Rightarrow s \in I_1$

- $\forall s, s' \cdot f(s) \mapsto f(s') \in K_2 \Rightarrow s \mapsto s' \in K_1$

To address the problem that a refinement mapping does not exist because the concrete specification hides the history or future details of the abstract specification, Abadi and Lamport (1991) proposed to use auxiliary variables such as history variables or prophecy variables with stuttering to create a valid refinement mapping.

2.3.2 Forward Simulations

Refinement mappings use trace equivalence to prove that some discrete system T_2 implements T_1. *Forward simulations* are generalizations of refinement mappings as they allow for relational relations between states in T_1 and T_2 instead of functions. Forward simulation is defined in Definition 3.16. Given the transition system T_1 and T_2, r is a forward simulation from T_1 to T_2 when $r[I_1] \subseteq I_2$ and for all transitions in T_1, there exists a transition in T_2 and the corresponding states are related with the relation r. Forward simulation, defined by Milner (1971), has been applied in cases such as data type refinements and distributed system refinements (Jifeng, 1989a; Josephs, 1988; Stark, 1984; Abrial, 2009).

Definition 2.3 (Forward Simulation)**.** Given transition systems $T_1 =< V_1, S_1, I_1, K_1 >$ and $T_2 =< V_2, S_2, I_2, K_2 >$, a forward simulation from T_1 to T_2 is a relation r from S_1 to S_2 that satisfies:

- $I_2 \subseteq r[I_1]$
- $\forall s, u, u' \cdot u \in r[s] \land u \mapsto u' \in K_2 \Rightarrow \exists s' \cdot s \mapsto s' \in K_1 \land u' \in r[s']$

2.4 Approaches to Modeling Time

2.4.1 Real-time Specification Patterns

The gap between informal requirements and formal specifications makes it challenging for the modelers to build models that describe the system behavior correctly. Dwyer et al. (1999) provided some specification patterns to guide developers in expressing system requirements directly in a formal specification language with model checkers. Figure 2.1 shows the specification patterns that could be used to describe the system behaviors. Occurrence patterns are used to represent the occurrence of the given events during system execution. For example, existence patterns require that events must eventually occur during system execution; absence patterns demand that the system execution is free of certain events; universality patterns require events to occur throughout the whole execution; bounded patterns require that the system execution contains convergent events. Instead of specifying the behavior of single events, order patterns describe the relative order between the given events during system execution. The precedence pattern illustrates the relationship between a pair of events where the occurrence of the first event enables the second event. The response pattern requires that an occurrence of the second event must follow the occurrence of the first event. Based on these specification patterns, Abid et al. (2012) extended the patterns to express real-time requirements in the design of real-time systems by using two kinds of *timing modifiers* that limits the time between events, namely **Within** and **Lasting**. The **Within** *timing modifier* is used to constrain the delay between two given events and the **Lasting** *timing modifier* is used to constrain the duration which some predicates hold for more than D time units.

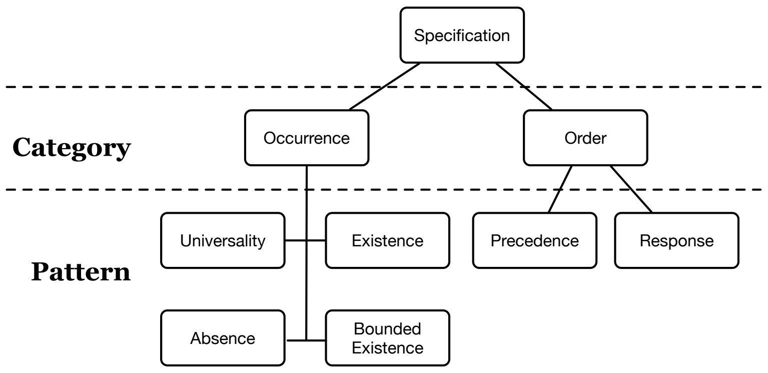

FIGURE 2.1: Specification Pattern Classification (Dwyer et al., 1999)

Dwyer et al.'s patterns are qualitative patterns rather than quantitative patterns because such patterns cannot be used to specify real-time properties. Based on these patterns, Konrad and Cheng (2005) have drawn attention to three kinds of real-time specifications and created mappings of real-time properties into three real-time temporal logics. Figure 2.2 shows these three categories of real-time specification pattern, namely duration ,periodic and real-time order. The minimum and maximum duration captures the minimum and maximum time a state formula should hold once it becomes true. The bounded recurrence pattern indicates that an event has to hold at least once within a bounded period of time. The bounded response pattern defines the deadline between two events and the bounded invariance specifies the delay between two events. Their real-time specification patterns contain templates for specifying time properties using three kinds of temporal logics, namely metric temporal logic (MTL) (Alur and Henzinger, 1990), timed computational tree logic (TCTL) (Alur, 1991) and real-time graphical interval logic (RTGIL) (Moser et al., 1997).

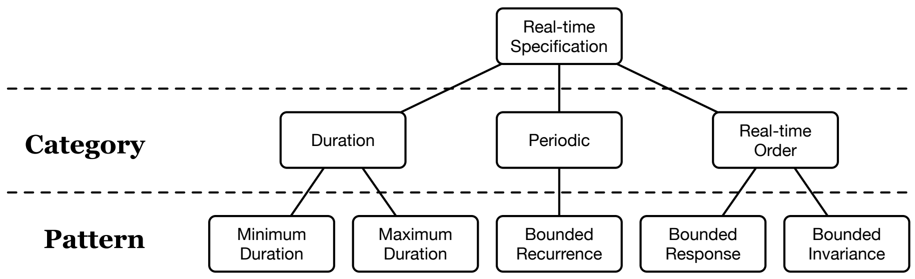

FIGURE 2.2: Real-time Specification Pattern Classification (Konrad and Cheng, 2005)

2.4.2 Temporal Logic of Actions

Based on the definition of transition systems in Definition 2.1, Henzinger et al. (1991) generalized the definition by associating minimal and maximum time delays of transitions. The Timed Transition System (TTS) is defined in Definition 2.4. On top of a transition system T^-, the TTS also has a minimum and a maximum delay for each transition. It is convenient to assume a discrete clock that records the timestamps of state changes (Alur and Henzinger, 1994). Besides, liveness of this digital clock needs to be guaranteed as time should always progress (Henzinger, 1991). Several specification languages are used to reason about the real-time properties of TTS. Alur and Henzinger (1991) translated the TTS to timed automata and used model checking techniques to check the properties of TTS. Alur and Henzinger (1994) defined Timed Propositional Temporal Logic (TPTL) that verifies timing properties of the timed state sequence in TTS.

Definition 2.4 (Timed Transition System (Henzinger et al., 1991))**.** A TTS $T =< V, S, I, K, l, u >$ consists of an underlying transition system $T^- =< V, S, I, K >$ as well as the minimal delay set l and a maximum delay set u. $l_k \in \mathbb{N}$ and $u_k \in \mathbb{N} \cup \{\infty\}$ is defined as the minimal and maximum delay for each transition $k \in K$.

Abadi and Lamport (1994) defined the semantics of Temporal Logic of Actions (TLA), which uses fairness properties to handle the requirements of liveness for the digital clock in real-time systems. TLA is structured in four tiers, namely constants with functions and predicates, state formulas for reasoning about the states, transition formulas for reasoning about the transition of states and temporal predicates for reasoning about the behavior traces (Abadi and Merz, 1996). The transition system can be specified with TLA in the form of Equation (2.1) (Méry and Poppleton, 2015).

$$\phi \triangleq Init_\phi \wedge \Box[Next]_f \wedge WF_f(N_1) \wedge SF_f(N_2) \tag{2.1}$$

In the formula, N_1 and N_2 denotes the system actions and $Next$ denotes the disjunction of all system actions regarding variables f. The WF and SF are weak fairness and strong fairness constrains on the actions N_1 and N_2. To model TTS with TLA, Abadi and Lamport (1994) define variable *now* with property that it never decreases. State functions are used to add volatile timers and persistent timers as lower-bound timers and upper-bound timers on actions. Besides the time constraints on actions, Abadi and Lamport (1994) also defined a liveness property that asserts that *now* gets arbitrarily large, which means that time can always progress. Fairness properties are specified to rule out the Zeno behavior in real-time systems. Their work put time bounds on individual actions by using *timers* to restrict the increase of the global clock. Based on their work, Zhang et al. (2010) specified time interval between two actions with TLA. The time specifications are expressed by the TLA+ language (Lamport, 2002), which is supported by the TLC model checker (Yu et al., 1999). However, the TLC model checker does not support parameterized specifications and refinement of time specifications.

2.4.3 Timed Automata

Timed automata is a formalism that can be used to model and analyze the timing behavior of real-time systems (Alur and Dill, 1994). In timed automata, systems are modeled as a state transition system where each state is related to clocks and the transition between states is also constrained by clocks. A timed transition table ξ is defined with a tuple $< \Sigma, S, I, C, E >$ (Alur and Dill, 1994), where

- Σ is a finite set of actions of ξ
- S is a finite set of states
- $I \subseteq S$ is a set of initial states
- C is a finite set of clocks
- $E \subseteq S \times S \times \Sigma \times 2^C \times \Phi(C)$ gives the set of transitions. The edge (s, s', a, r, g) from E is a transition from state s to s' with the input symbol a, clock resets r and clock constraint g over C. The transition from state s to s' is constrained by the clock constraint g and input symbol a. Here the set $\Phi(C)$ of clock constraints g is defined inductively by Formula 2.2, where x is a clock in C and c is a constant.

$$g := x \leq c \| c \leq x \| \neg g \| g_1 \wedge g_2, \tag{2.2}$$

Timed automata are supported by model-checking tools such as UPPAAL (Larsen et al., 1997), KRONOS (Yovine, 1997) and PRISM (Hinton et al., 2006). UPPAAL is a toolbox designed to verify systems that can be modelled as networks of timed automata extended with integer variables, structured data types and channel synchronization (Behrmann et al., 2004). It is usually used to model real-time systems. However, many real-life systems also exhibit stochastic behavior such as component failures and unreliable communication. So probabilistic verification tools such as PRISM are used to analyze systems with stochastic behavior (Norman et al., 2012).

The state explosion problem is widely agreed to be a major challenge in applying model checking to large and complex real-world systems (Rozier, 2011). Verification of real-time system by model checking is more challenging as it introduces time. So an abstraction refinement procedure is needed when modelling timed systems. However, some widely used formalism to model real-time systems such as timed automata do not support refinement of the model, which means that all the details have to be included when designing the model. Dierks et al. (2007) used the idea of Counter-Example Guided Abstraction Refinement (CEGAR) (Clarke et al., 2000) to automate all the steps of the abstraction refinement loop in timed automata. Figure 2.3 shows a fully automated abstraction refinement loop, which starts with an abstraction model. After performing model checking some counter examples are found. Then the counterexample analyzer is used to check what is missing in the abstract model that leads to the counter example. And a refinement is performed based on the the result of analyzer to get rid of the counter examples. The termination of this abstraction refinement loop is guaranteed as each iteration removes at least one of the counter examples. In the CEGAR loop, if an abstract model satisfies the given specifications, the concrete also satisfies them.

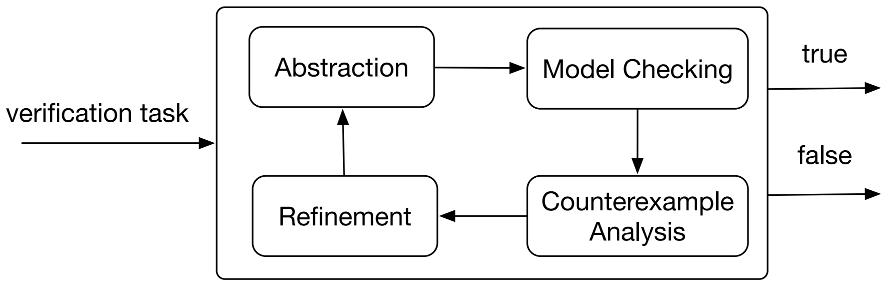

FIGURE 2.3: Counter-Example Guided Abstraction Refinement Loop (Dierks et al., 2007)

CEGAR has already been used in refinement based modeling of complex CPS systems such as heart and pacemaker modelling (Jiang et al., 2013). The heart is modeled using timed automata based on the physiology of heart, which is abstracted with time properties. And a CEGAR framework is adapted to refine the heart model to get rid of spurious counter-examples found by UPPAAL model checker.

2.4.4 Time Modeling in Event-B

Event-B (Abrial, 2009; Event B.org, 2019a) is a formal modelling method based on set theory and first-order logic, which is usually used for system-level modelling and analysis with refinement and reasoning on the model (Event B.org, 2019a). Refinement is used in Event-B to specify a complex system with different abstraction levels to lower the difficulties to model a sophisticated system in a step-wise way. To verify the consistency between refinement levels, mathematical proofs are used to do the verification.

2.4.4.1 Event-B Syntax

A Event-B model that uses mathematical theory to describe a discrete transition system usually contains two kind of components: *contexts* and *machines*. *Contexts* specify the static part of a model whereas *machines* specify behavioral properties of Event-B models (Hoang, 2010).

A *context* may contain carrier sets, constants, axioms, and theorems. Carrier sets are used to define types used in the model. Carrier sets and constants are constrained by axioms and theorems that are derived from axioms. A *machine* may contain variables, invariants and events. Variables denote the properties of a state of the *machine*. These variables are constrained by invariants $I(v)$, where v denotes the constrained variables. The events describe the changes of the states in a *machine*. The invariants should be preserved by all the events of a machine. The most general form of an event is shown in (2.3) below. Each event consists of a guard $G(p, v)$ as well as a before-after predicate $BA(p, v, v')$. $G(p, v)$ defines the enabledness of the event with the parameter p. If several events are enabled, at most one of them would be chosen non-deterministically to occur. $BA(p, v, v')$ describes the changes to variables upon event execution where v' stands for the variables after the execution of the event.

$$\text{evt} \triangleq \ \text{where } G(p, v) \text{ then } v :| BA(p, v, v') \text{ end} \tag{2.3}$$

2.4.4.2 Abstraction and Refinement

Formal modelling is used to address the problem of lack of precision of specifications. However, it does not handle the problem of complex requirements and specifications (Butler, 2013a). Using abstraction that focuses on the general goal of the system instead of those complex details all in one level, makes it easier to model and reason about the system. However, the complexity of the system is not ignored. Instead, after the verification of the abstract model, several refinement steps can be carried out. Each step adds details to the previous step based on the verification proof of the previous one and proof obligations are proved to verify these refinement steps.

There are two types of refinement for Event-B models. The process that only adds features to the system by introducing more concrete elements is called superposition refinement (Abrial, 2009). And the process that transforms some states and events of the model is called data refinement. Superposition refinement is usually used for system specifications while data refinement is used to ease the implementation of the model (Sarshogh, 2013). The variables in the refined machine are linked to the abstract state variable by using gluing invariants to ensure the consistency of the system.

2.4.4.3 Convergent Events

The Event-B notion of convergence requires convergent events to become disabled eventually without any fairness assumptions. In Event-B models, not all events are expected to be convergent. In system modeling, the modeler defines events that must not execute forever to be convergent. The *convergent* event must decrease a variant in Event-B models, whereas *anticipated* events must not increase the variant. The *anticipated* events must become *convergent* at some stage of refinement.

Definition 2.5 (Convergent Events)**.** A group of events A is defined to be convergent when

- $V(v)$ is an integer expression.
- Given invariants $I(v)$ and the group of events A is enabled, the variant $V(v)$ is a natural number.
$$I(v) \wedge G_A(v) \Rightarrow V(v) \in \mathbb{N}$$
- An execution of each event $e \in A$ decreases $V(v)$.

$$I(v) \wedge G_e(v) \wedge BA_e(v, v') \Rightarrow V(v') < V(v)$$

2.4.4.4 Proof Obligation

In Event-B, Proof Obligations (PO) are used to specify the consistency between different refinement levels (Abrial, 2009). A brief description of the main PO rules is described in Table 2.1. These POs are automatically generated by the Rodin tool.

2.4.4.5 Rodin: An Event-B Modelling Tool

Rodin (Event B.org, 2019b) is an Eclipse-based integrated development environment (IDE) for Event-B, which supports construction and verification of Event-B models (Abrial et al., 2010). Engineers can use it to find specification errors of a system by checking proof obligation violations of the model.

The Rodin tool chain contains three components: the static checker (SC), the proof obligation generator (POG) and the proof obligation manager (POM) (Abrial et al., 2010). The workflow of Rodin is shown in Figure 2.4. SC checks the syntax errors of the *machine* and *context* and provides feedback to the user. After the check POG generates POs such as feasibility, refinement and convergence for the model. Some POs can be discharged automatically, while some POs have to be discharged manually by using some interactive prover such as Atelier B (ClearSy, 2019). And POM keeps track of these POs and associate proofs with obligations by constructing proof trees.

TABLE 2.1: Event-B Proof Obligation Rules (Abrial, 2009)

Proof Obligation Rule	**Description**
Invariant Preservation(INV)	Ensures that each invariant in a machine is preserved by each event.
Feasibility(FIS)	Ensure that a non-deterministic action is feasible.
Guard Strengthening(GRD)	Ensure that the concrete guards are stronger than their corresponding abstract ones.
Simulation(SIM)	Ensure that each action in an abstract event is correctly simulated in the corresponding refinement
Numeric Variant(NAT)	Ensure that a proposed numeric variant is a natural number under the guards of each convergent or anticipated event.
Numeric Variant(VAR)	Ensure that each convergent event decreases the proposed numeric variant
Finite Set Variant(FIN)	Ensure that a proposed set variant is a finite set under the guards of each convergent or anticipated event.
Theorem(THM)	Ensure that a proposed context or machine theorem is provable.
Well-definedness(WD)	Ensure all elements of the model are well defined.

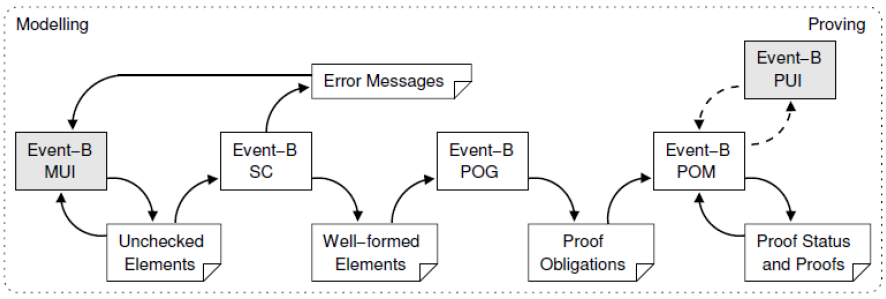

FIGURE 2.4: Tool-Chain in Event-B Core and The User Interface (Abrial et al., 2010)

The graphical user interface of Rodin consists of two parts: user interface for modelling (MUI) and user interface for proving (PUI). Figure 2.4 shows the tool chain that makes up the core of Rodin (Abrial et al., 2010). The PUI can be used to do interactive proofs. Engineers can use the PUI to prune the proof tree or add new hypotheses and apply tactics on the hypothesis and goal. MUI can be used to edit the *machine* and *context* in the model.

2.4.4.6 Extend Event-B with Discrete Timing Properties

Event-B is a general purpose modeling language that supports model refinement but lacks explicit support for expressing and verifying timing constraints (Sulskus et al., 2016). Several studies have developed patterns to extend Event-B models with real-time properties.

Cansell et al. (2007) developed an action-reaction pattern to model the causal order between events. Time constraints are imposed between action and reaction events to model real-time properties. They use variable *clock* to denote the current time, and variable *at* to denote a set of timestamps of future activated events. They use a *post* event to create a new timestamp of the activated event in the timestamp set. The *tick* event takes a new value of a time in the future that is earlier than any of the active timestamps and assigns it to the current time. When the current time reaches the active timestamp, the *process* event removes the current time from the active timestamp set. However, refinement patterns are not provided by this approach. Also, this pattern lacks the ability to model the interrupt behavior between the trigger and response events.

Abrial et al. (1996) proposed an approach to modeling and refining timing properties in classical B, which adds a clock variable representing the current time and an operation which advances the clock (Butler and Falampin, 2002). The approach is influenced by Lamport's work (Abadi and Lamport, 1994) and ensures the timing properties are satisfied by preventing behaviors in which the clock advances to a point where deadlines would be violated. Based on this approach, Sarshogh and Sulskus added explicit support for trigger-response properties with deadline, delay, expiry and interval timing properties (Sarshogh and Butler, 2011; Sulskus et al., 2016).

To formally model the timing properties for the trigger-response pattern in control systems, Sarshogh and Butler (2011) proposed an approach that categorizes timing constraints in three groups: delay, expiry and deadline, which are denoted in (2.4a), (2.4b) and (2.4c) below. All these three timing constraints follow a trigger-response pattern where trigger and response events are modeled as events in Event-B. Specification (2.4a) shows that the *Response* event can only happen if the *delay* period has passed following the occurrence of the *Trigger* event. Specification (2.4b) shows that if the *expiry* period has passed then the *Response* event can never happen. Specification (2.4c) denotes that if the *Trigger* event occurs, then one of the events $Response_1$ to $Response_n$ must occur before *deadline* passes. To model these three timing properties in Event-B, a global clock as well as a tick event are added to model the discrete time.

$$Delay(Trigger, Response, delay) \tag{2.4a}$$

$$Expiry(Trigger, Response, expiry) \tag{2.4b}$$

$$Deadline(Trigger, Response_1 \vee .. \vee Response_n, deadline) \tag{2.4c}$$

Figure 2.5 shows the trigger response pattern as an Event-B machine for the delay and deadline constraints, where *trigger* and *response* are modelled as events. The response event *response* must occur within time *ddl* of trigger event *trigger* occurring and can only occur if the delay period has passed. We use tT to refer to the time that a trigger event happens, and we use tR to refer to the time that a response event happens. $Ga(v)$, Act_a and $Gb(v)$, Act_b are the guards and actions of untimed trigger and response events respectively. Invariant $@inv1$ and $@inv2$ specify the delay and deadline timing property between *trigger* and *response* respectively. Guard $@grd3$ of the *response* event guarantees that the response is disabled when the global clock has not passed the delay period thus preserving $@inv1$. Guard $@grd1$ of the *Tick* event constrains the global clock not to tick when the response event is missing its deadline thus preserving $@inv2$. $@inv3$ is needed to prove invariant $@inv2$. When modeling the expiry time constraint, $@grd3$ of *response* event should be $clk < tT + expiry$ to guarantee that the response is disabled when the global clock has passed the expiry period.

```
invariants
  @inv1 tT<tR⇒tR−tT≥ dly
  @inv2 tR≥ tT⇒tR−tT≤ ddl
  @inv3 t=TRUE∧r=FALSE⇒clk−tT≤ ddl
event trigger
  where
    @grd1  t=FALSE
    Ga(v)
  then
    @act1 t:= TRUE
    @act2 tT:= clk
    Act_a
end
```

```
event response
  where
    @grd1 t=TRUE
    @grd2 r=FALSE
    @grd3 clk≥ tT+dly
    Gb(v)
  then
    @act1 r:= TRUE
    @act2 tR:= clk
    Act_b
end

event Tick
  where
    @grd1 t=TRUE∧r=FALSE⇒clk+1−tT≤ ddl
  then
    clk:=  clk+1
end
```

FIGURE 2.5: Model Timing Properties of Trigger-Response Events with Delay and Deadline

Sarshogh's approach handles the system with trigger and response properties without specifying any possible interrupt events from the environment. To model a more complex system that supports interrupt events in interrupting current time intervals, Sulskus et al. (2016) introduced the notion of time interval constraints in Event-B, which is a higher-level abstraction in terms of state machine.

$$Interval(T_1[, ..., T_i]; R_1[, ..., R_j]; [I_1, ..., I_k]; TP_1(t_1)[, TP_2(t_2)]) \tag{2.5}$$

The notation of the time interval constraint is presented in (2.5), and consists of three kinds of events: trigger events: $T \in T_1..T_i$; response events: $R \in R_1..R_j$ and interrupt events: $I \in I_1..I_k$. Each pair of trigger and response events composes an active interval instance, which is constrained by the timing property $TP(t)$ of duration t. Here TP stands for deadlines, delays and expiries. The interval can have one of five TP configurations: (i.) Deadline; (ii.) Delays; (iii.) Expiry; (iv.) Delay and Expiry; (v.) Delay and Deadline (Sulskus et al., 2016). The interrupt event always interrupts an active interval instance but does not block it if no active interval instance exists. When a trigger event happens, an active interval instance is created. The response event is constrained by one or more timing properties. An interrupt event can be executed at any given time regardless of the state the model is in.

2.4.4.7 Refinement Patterns of Timing Properties

There are several patterns developed by Sarshogh to refine deadlines, delay and expiry (Sarshogh and Butler, 2011). As shown in Figure 2.6, to refine an abstract deadline D between the trigger-response pair to sequential sub-deadlines $D_1..D_n$, there should be invariants to ensure the order of sequential sub-deadlines. Also, the sum of the duration of sub-deadlines should be less than the abstract deadline duration. To refine an abstract deadline to alternative sub-deadlines, a gluing invariant that indicates that the abstract event is equivalent to the occurrence of alternative sub-events is required to verify the consistency between refinement levels. To refine alternative sub-deadlines with sequential sub-deadlines and expiries, the sum of sequential sub-deadline durations should be less than the abstract deadline duration.

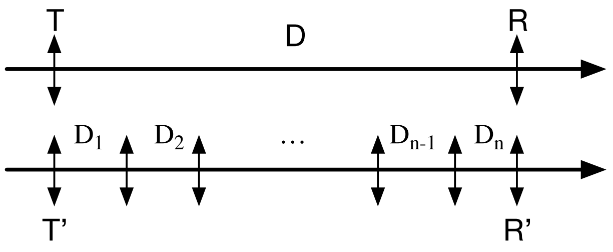

FIGURE 2.6: Refine an abstract deadline D to sequential sub-deadlines $D_1..D_n$ (Sarshogh and Butler, 2011)

Sulskus (2017) concluded three refinement transformations for the interval timing property, namely alternative interval transformation, sub-interval transformation and abort to response transformation.

The alternative transformation refines an abstract interval property into several alternative timing interval properties. The alternative intervals should share the same trigger event, response event and timing properties. Additional gluing invariants are needed to verify that the concrete interval properties are consistent with the abstract time interval property.

The sub-interval transformation refines an abstract interval into a sequence of sub-intervals. The sequential sub-intervals could be connected with a transition event which is a response event of the preceding interval instance and a trigger event of the succeeding one. Additional gluing invariants are needed to guarantee that the sum of the delay and deadline duration of sub-intervals is consistent with the abstract delay and deadline duration.

The abort-to-response transformation refines the abort event of the abstract interval into a response event. Moreover, new interrupt events could be introduced in refinement steps. The transformation also requires that the concrete timing property between trigger and response events is consistent with the abstract timing property.

2.5 Fairness Modeling

2.5.1 Fairness

In concurrent systems, fairness is an important concept that constrains the selection of events for execution. Francez (1986) defined fairness as a restriction on some infinite behavior according to eventual occurrence of some events. Fairness guarantees that every process, represented by some subset of events, gets a chance to make progress, regardless of what other processes do. There are two typical fairness assumptions: *strong* fairness and *weak* fairness assumption. *Strong* fairness requires that a process that is infinitely often enabled should be executed infinitely often while *weak* fairness requires a process that is continuously enabled should be executed eventually.

2.5.2 Bounded Fairness

The fairness assumptions guarantee an eventual occurrence of an event but provide no bound on how soon the event will occur. However this qualitative specification and analysis are inadequate for many systems and applications (Pnueli and Harel, 1988). For example, some process control systems may require that any process p_i that wish to request the resource be granted the resource within the next k times that other processes arriving after p_i are granted with the resource. In this situation, the system with fairness assumptions cannot meet the requirement. Dershowitz et al. (2003) proposed bounded fairness ($k-fairness$) to specify and reason about the frequency of occurrence of events, which guarantees occurrence of an event within a fixed number of occurrences of another event.

2.5.3 Finitary Fairness

Bounded fairness specifies fairness of a particular event relative to another, while the interleaving model of concurrency represents the concurrent execution of several independent events, where bounded fairness is not applicable in this case. Alur and Henzinger (1998) proposed the definition of finitary fairness which imposes bounds on the relative frequency in scheduling of a set of events. Based on their definition, a set of transitions F in an LTS is weakly $k-\mathit{bounded}$ iff for all transitions τ of F, the transition τ cannot be enabled for more than k consecutive positions without being taken.

Sekerinski and Zhang (2013) proposed a method for expressing finitary weak fairness in Event-B. They define a fair event system as follows:

Definition 2.6. A fair event system P is a structure (Q, I, F, E, T) where

- Q is the set of states,
- I is the set of initial states, $I \subseteq Q$,
- F is the number of fair events,
- E is the set of all events with $\mathit{card}(E) \geq F$,
- T is the set of transitions, relations over $Q \times Q$ indexed by E

Based on the fair event system, they give the definition of the finitary weakly fair transformation, which expresses finitary fairness by introducing counter variables $C_1, ..., C_F$ for each fair event. The counter C_i of event e_i for $i \in 1..F$ indicates that e_i must be taken or disabled at least once in the next C_i transitions. They also introduce a permutation p which satisfies $\forall j \cdot j \in 1..F \Rightarrow C_{p(j)} \geq j$ to guarantee that only one counter can be 1. On every transition, the guards of all fair events must be tested: if a fair event is enabled but not taken, its counter must be decreased by 1, otherwise the counter is reset to some finite value. However, this stating the existence of a permutation will increase exponentially with the number of fair events (Sekerinski and Zhang, 2013).

2.6 Real-time Scheduling

Figure 2.7 shows a typical interaction pattern between the scheduler and different tasks (Alur, 2015). Initially the task is in the *idle* mode. When the task needs the processor to do some processing, it sends an event to the scheduler for processing and changes to the *wait* mode. When the scheduler decides to allocate the processor to the task, it notifies the task using the event *run* and the task changes its mode to *running*. After the task finishes its execution and sends a *done* event to scheduler and changes its mode to idle. During the running of a task, if the processor allows another task to preempt the current one, then the scheduler sends a preempt event to the task and the task changes from the running mode to the wait mode. This scheduling is called a preemptive scheduler. If the scheduler does not allow one task to preempt another, then this is a non-preemptive scheduler. Most real-time scheduling systems are preemptive systems which allow tasks with high priorities to preempt tasks with lower priorities. This allows the tasks with high priorities to finish first. There are two kinds of scheduling policies of preemptive scheduling system for independent tasks, one is static priority scheduling, another is dynamic priority scheduling. There is also a scheduling policy for dependent tasks that shared the same resource. All of these three policies are described in the following section.

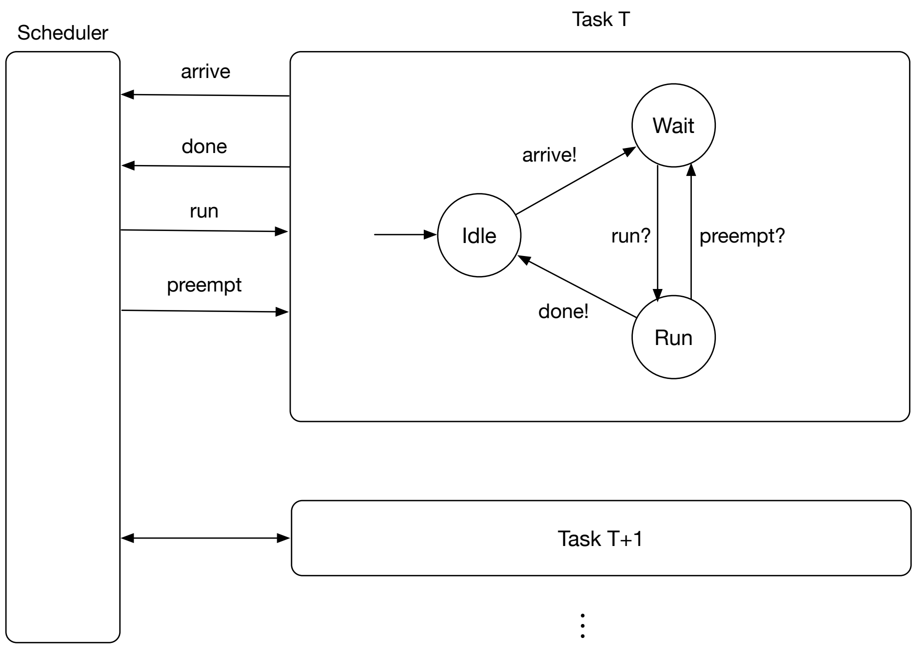

FIGURE 2.7: Interaction between the Scheduler and the Tasks (Alur, 2015)

2.6.1 Static Priority Scheduling For Independent Tasks

For a static priority scheduling policy, the tasks are assigned with fixed priorities, and whenever the scheduler has to make a choice, the task with the highest priority is chosen (Alur, 2015). Rate-Monotonic (RM) is one of the static priority scheduling policies that schedule a set of periodic independent hard real-time tasks in a system with a single CPU (Liu and Layland, 1973). The RM policy assumes that the deadline of each task is equal to its period. It assigns static priorities based on the task periods. The tasks with the shortest period gets the highest priority and vice versa. Schedulers choose the task with highest priority to execute.

2.6.2 Dynamic Priority Scheduling For Independent Tasks

A dynamic priority scheduling policy may assign different priorities to the same task at different times. Earliest Deadline First (EDF) is one of the dynamic priority scheduling policies, which also schedules a set of periodic independent tasks. However, the task whose deadline is going to expire first is granted the highest priority. In a periodic job model, each job is active at different times, so the value of a deadline depends on the time t. The priority of the same task at different timestamp t may therefore be different. The scheduler will choose the task with the highest priority to execute.

2.6.3 Scheduling Dependent Tasks

Concurrently executing tasks must exchange information and access common data resources in order to cooperate to achieve the overall system objectives, so scheduling dependent tasks is also important in distributed real-time systems (Kopetz, 2011). When the task with high priority wants to gain access to a resource which is already acquired by a task with lower priority, it cannot preempt the lower priority one as this would cause deadlock. Instead, it should wait for the task with lower priority to release the resource. A solution developed by Sha uses a priority inheritance protocol to prevent deadlocks (Sha et al., 1990). The rule of the priority inheritance protocol is to let the task with lower priorities inherit the highest priority of tasks blocked by the resource it is acquiring. Then the task with higher priority will not preempt the lower one and no deadlock will occur.

2.7 Conclusion

In conclusion, formal methods help to verify functional and timing requirements in CPS with tool support. The TLA approach mainly focuses on reasoning about time bounds on individual actions. Timed automata produce a finite-state space model to reason about timing properties in system design. However, refinement is not supported natively by the UPPAAL model checker. The complexity of system design has to be tackled all at once. The Event-B formalism uses the deductive verification approach to reason about timing constraints between different actions in a real-time system. Refinement could be used to develop the system in a stepwise manner. An unbounded variable is used to model time, which presents the natural property that time could always progress.

However, existing work lacks formal treatment of the relation between trigger events, intermediate events and response events in time modeling. Infeasible responses will lead to the *Zeno* behavior whereby an infinite number of events occur within a bounded period of time. Thus additional conditions are required for the correctness of a real-time system. Forward simulations could be used to reason about the consistency between different refinement levels regarding infinite traces.

Chapter 3

Trace Semantics and Refinement Patterns for Real-Time Trigger-Response Properties

As discussed in Section 2.7, existing work lacks a proper treatment of key issues in timed systems, namely divergence of new events in the refined machine and infeasible occurrence of events without fairness assumptions. In our setting, the global clock is constrained by the response events based on the syntax of deadline time constraints. Infeasible responses will lead to the *Zeno* behavior whereby an infinite number of events occur within a bounded period of time. In this chapter we address these problems with proof obligations and fairness assumptions on intermediate events and response events. We provide trace semantics for a trigger-response property. We explore sufficient conditions under which all the traces of an Event-B model satisfy a trigger-response property, in the form of Event-B proof obligations and fairness assumptions. Also, we explore sufficient conditions under which the refinement is valid in terms of infinite traces. Fairness assumptions, relative deadlock freedom, and conditional convergence are adopted to refine the discrete timed models. We present infinite state trace refinement with gluing relations between abstract and concrete state traces. The definition of generic refinement that allows relabelling and stuttering events is presented with event trace inclusion. We summarize infinite trace properties as well as sufficient conditions under which timing properties are preserved by refinement.

3.1 Trace Semantics and Properties of Event-B Models

3.1.1 Trace Semantics of Event-B Machine

Event-B is a modeling approach for formalizing discrete transition systems. We abstract away from the concrete syntax of Event-B and treat a machine as a form of labeled transition system (Definition 3.1). In the definition, the machine consists of state set S and initial states $init \subseteq S$. We use E to denote a set of event labels of the machine M. The transition relation K is used as a function from event label set E to the relation between states.

Definition 3.1 (Machine)**.** A machine M is a tuple $< S, init, E, K >$ consisting of

- S: a set of states, each state s is a mapping of variables of M to their values;
- $init$: a set of initial states $init \subseteq S \wedge init \neq \varnothing$, which correspond to initial configurations;
- E: a set of event labels of the machine M;
- K: a transition relation $K \in E \rightarrow (S \leftrightarrow S)$ that relates pairs of states;

Given the definition of a *machine*, we define $traces(M)$ of machine M to describe the infinite behavior of the system in terms of the set of pairs of state trace and event trace (u_s, u_e), which could be presented as an infinite sequences of alternating states and events of the form $< u_s(0), u_e(0), u_s(1), u_e(1), ... >$ in Definition 3.2.

Definition 3.2 (Infinite Trace of a Machine)**.** The set of infinite traces $traces(M)$ of a machine M is defined as:

$$\begin{aligned} traces(M) \triangleq \{(u_s, u_e) \mid\ & u_s \in \mathbb{N} \rightarrow S \wedge u_e \in \mathbb{N} \rightarrow E \wedge u_s(0) \in init \\ & \wedge \forall i \cdot i \geq 0 \Rightarrow u_s(i) \mapsto u_s(i+1) \in K(u_e(i))\} \end{aligned}$$

The event traces (Definition 3.3) and state traces (Definition 3.4) of a machine are defined below; their properties are studied in the next section.

Definition 3.3 (The Set of Event Traces of M)**.** A set of event traces of M is defined as:

$$e\_traces(M) = \{u_e \mid \exists u_s \cdot (u_s, u_e) \in traces(M)\}$$

Definition 3.4 (The Set of State Traces of M)**.** A set of state traces of M for some event trace u_e is defined as:

$$s\_traces(M, u_e) = \{u_s \mid (u_s, u_e) \in traces(M)\}$$

Lemma 3.5 shows that the state trace set of an event trace in machine M is not empty.

Lemma 3.5.

$$u_e \in e\_traces(M) \Leftrightarrow s\_traces(M, u_e) \neq \varnothing$$

3.1.2 Trace Properties

We adopt weak fairness assumptions on infinite traces to exclude traces with *Zeno* behaviors, whereby an infinite number of events occur within a finite period. In Definition 3.6, we extend the machine definition in Definition 3.1 with a set of events $F \subseteq E$. The traces of a machine that is weakly fair on the event set F satisfy the property that if F is continuously enabled from some point, then $f \in F$ will eventually occur on the trace. A machine is defined to weakly fair on F when $\forall (u_s, u_e) \in traces(M) \Rightarrow WFair((u_s, u_e), F)$.

Definition 3.6 (Machine Traces with Weak Fairness)**.** A machine M with weak fairness is a tuple $< S, init, E, K, F >$ consisting of a machine $< S, init, E, K >$ and a set of weakly fair events $F \subseteq E$. The trace $(u_s, u_e) \in traces(M)$ is defined as weakly fair on F as $WFair((u_s, u_e), F)$, formally:

$$\begin{aligned} & WFair((u_s, u_e), F) \triangleq \\ & \forall i \cdot i \geq 0 \wedge u_s(i) \in dom(K(F)) \Rightarrow \exists j \cdot j \geq i \wedge (u_e(j) \in F \vee u_s(j) \notin dom(K[F]) \end{aligned}$$

The normal Event-B notion of convergence requires convergent events to become disabled eventually without any fairness assumptions. Our work extended the notion of convergence to conditional convergence to define a set of events are converging under certain condition $Q(v)$ in Definition 3.7. The notion of conditional convergence is a generalized version of convergence as it only requires the events to be convergent under some specific condition. If the condition is not satisfied, then the events do not need to converge.

Definition 3.7 (Conditional Convergence)**.** A group of events A is defined to be conditional convergent under the condition Q when

- $V(v)$ is an integer expression.
- When the group of events is enabled, the variant $V(v)$ is a natural number.

$$I(v) \wedge G_A(v) \Rightarrow V(v) \in \mathbb{N}$$

- An execution of each event $e \in A$ decreases $V(v)$ provided $Q(v)$ holds.

$$I(v) \wedge G_e(v) \wedge S_e(v, v') \wedge Q(v) \Rightarrow V(v') < V(v)$$

Based on Definition 3.7, we provide the conditional convergence trace property in Definition 3.8 such that event set A must not be executed forever under the condition Q.

Definition 3.8 (Conditional Convergence Property)**.** Given that machine M is weakly fair towards A, trace $(u_s, u_e) \in \mathit{traces}(M)$ is defined to be convergent as $(u_s, u_e) \models \mathit{cov}(A, Q)$, formally:

$$(u_s, u_e) \models \mathit{cov}(A, Q) \triangleq \forall i \cdot (\exists j \cdot j \geq i \Rightarrow (u_s(j) \notin \mathit{dom}(K(A)) \vee \neg Q))$$

Abid et al. (2013) proposed a set of specification patterns that can be used to express real-time requirements in the design of reactive systems, namely existence patterns, absence patterns, and response patterns. Existence patterns are used to express that certain events must occur in every trace of the system; absence patterns require that certain events should never occur in every trace of the system. A trigger-response pair in Event-B models can be used in these patterns. Details of modeling real-time specification patterns with trigger-response properties are provided in Section 6.1. In the trigger-response pair, the trigger event is followed by its possible response events eventually. Given a machine with events E, a trigger-response pair has the form (T, R) where $T \subseteq E \wedge R \subseteq E \wedge T \cap R = \varnothing$. We define when an event trace of a machine satisfies the trigger-response property $\mathit{TR}(T, R)$ as Definition 3.9. An event trace u_e satisfies $\mathit{TR}(T, R)$ provided that any occurrence of a trigger event $t \in T$ is followed eventually by a response event $r \in R$ and the trigger event does not recur within the trigger-response pair to avoid the recurring delay of response event.

Definition 3.9 (Trigger-Response Property)**.** Given the trigger-response pair (T, R) that satisfies $T \subseteq E \wedge R \subseteq E \wedge T \cap R = \varnothing \wedge T \neq \varnothing \wedge R \neq \varnothing$, trace $u_e \in \mathbb{N} \to E$ is defined to satisfy the trigger-response property $\mathit{TR}(T, R)$ as:

$$\begin{aligned}&u_e \models \mathit{TR}(T, R) \triangleq \\ &\forall i \cdot i \geq 0 \wedge u_e(i) \in T \Rightarrow (\exists j \cdot j > i \wedge u_e(j) \in R \wedge \forall k \cdot i < k < j \Rightarrow u_e(k) \notin T)\end{aligned}$$

Given a machine M with event labels E that represent the names of the events in M, we provide the syntax of real-time trigger-response properties as Definition 3.10 based on a trigger-response pair (T, R), where $T \subseteq E$ are trigger events, $R \subseteq E$ are response events, and $T \cap R = \varnothing$. We use $\mathit{Resp}(T, R, w, d)$ to denote the timing property between trigger events T and response events R, where w is the delay constraint and d is the deadline constraint.

Definition 3.10 (Real-Time Trigger-Response Property)**.** A real-time trigger-response property $\mathit{Resp}(T, R, w, d)$ of a machine M with event labels E consists of:

- trigger events $T \subseteq E \wedge T \neq \varnothing$;

- response events $R \subseteq E \wedge R \neq \varnothing \wedge T \cap R = \varnothing$;
- a delay $w \in \mathbb{N}$ and a deadline $d \in \mathbb{N}$.

To model real-time systems, we introduce a special event *Tick* in event traces to represent the progress of time. In a real-time system, one essential property requires that time should always progress. The event traces of real-time systems should always be *infinite traces* with *infinitely many Tick* events. Definition 3.11 provides the timed property of a timed trace, which guarantees that there is an infinite number of *Tick* events occurring in the event trace. Moreover, only a finite number of $non - Tick$ events could occur between any two *Tick* events.

Definition 3.11 (Timed Trace)**.** Event trace u_e is defined to be a timed trace as

$$timed(u_e) \triangleq \forall i \cdot i \in \mathbb{N} \Rightarrow (\exists j \cdot j \geq i \wedge u(j) = Tick)$$

Given a trigger-response pair (T, R), we use the number of *Tick* events between the trigger event T and response event R to denote the corresponding timing property for the trigger-response pair. Definition 3.12 shows the bounded real-time property for the trigger-response pair (T, R). The number of *Tick* events between trigger and response events is restricted by a lower bound w and an upper bound d. The response event R must occur within time d of trigger event T occurring and can only occur if the delay period w has passed. Given the event trace u_e, the *projection* $u_e \upharpoonright A$ is the event trace restricted to only those events in A. The length operator $\#(u_e)$ defines the length of a finite event trace or the finite sub-trace of an infinite trace. The sub-trace function $u_e[i, j]$ defines the sub-trace between index i and j as $< u_e(i), u_e(i+1), ..., u_e(j) >$. The event trace that satisfies the bounded real-time property should satisfy the trigger-response property as well as the timed property.

Definition 3.12 (Bounded Timing Property)**.** An infinite event trace u_e is said to satisfy the bounded timing property $u_e \models Resp(T, R, w, d)$ provided the following conditions hold:

$$\begin{aligned} &u_e \models Resp(T, R, w, d) \triangleq timed(u_e) \wedge \forall i \cdot i \geq 0 \wedge u_e(i) \in T \Rightarrow \\ &\quad \exists j \cdot j > i \wedge u_e(j) \in R \wedge \forall k \cdot i < k < j \Rightarrow u_e(k) \notin T \wedge (w \leq \#(u_e[i, j] \upharpoonright Tick) \leq d) \end{aligned}$$

We define the traces of Event-B models with bounded timing properties in Definition 3.13. Here we use $M \wedge Resp(T, R, w, d)$ to present a machine M that is constrained to satisfy timing properties $Resp(T, R, w, d)$.

Definition 3.13 (Machine with Bounded Timing Property)**.**

$$traces(M \wedge Resp(T, R, w, d)) \triangleq \{(u_s, u_e) \mid (u_s, u_e) \in traces(M) \wedge u_e \models Resp(T, R, w, d)\}$$

3.2 Enforcing Real-time Trigger-Response Properties in Event-B Models

In a trigger-response pair, the response event is not necessarily enabled after the execution of a trigger event. Instead, there may be a group of intermediate events H that is enabled by a trigger event $t \in T$ and whose execution leads to a response event $r \in R$ being enabled. In this section we explore sufficient conditions under which a behavioral trace satisfies the trigger-response property. For each trigger and response pair (T, R), a set of intermediate events $H \subset E$ s.t. $T \cap H = \varnothing$ and $R \cap H = \varnothing$ is assumed. This set is chosen by the modeler and consists of events that are initiated by a trigger event and whose execution is intended to lead to a response being enabled. The intermediate events should converge towards a response event being enabled and should not be disabled unless the response event is enabled. Weak fairness assumptions on the intermediate events guarantee that the intermediate events get executed sufficiently often to lead to the response event being enabled. We require that if a response event is enabled, it cannot be disabled by any event other than a response event to avoid the scenario that intermediate events and response events are enabled alternatively but never get executed under the weak fairness assumption. Weak fairness assumptions on the response events guarantee that an enabled response event eventually gets executed.

Theorem 3.14 provides additional conditions for the machine to satisfy the trigger-response property $TR(T, R)$.

Theorem 3.14. *Let M be an Event-B machine, let $TR(T, R)$ be a trigger-response property and let $H \subset E$ such that $T \cap H = \varnothing$ and $R \cap H = \varnothing$. If the following conditions 1)-5) are true, then $traces(M)$ satisfies $TR(T, R)$ when M is weakly fair on H and R:*

1. $G_R(v) \Rightarrow \neg G_T(v)$;
2. $I(v) \wedge G_t(v) \wedge S_t(v, v') \Rightarrow \neg G_t(v') \wedge (G_H(v') \vee G_R(v'))$;
3. $e \in E \setminus (T \cup R) \wedge I(v) \wedge G_e(v) \wedge S_e(v, v') \wedge G_H(v) \Rightarrow (G_H(v') \vee G_R(v'))$;
4. $e \in E \setminus (T \cup R) \wedge I(v) \wedge G_e(v) \wedge S_e(v, v') \wedge G_R(v) \Rightarrow G_R(v')$;
5. *The intermediate events H are conditional convergent under the condition that response events R are disabled;*

Proof. Let $u \in traces(M)$ and assume $u_e(i) \in T$ but $\forall j \cdot j \geq i \wedge u_e(j) \notin R$. Then by weak fairness of R, R is not continuously enabled at any point after i. That is, R is infinitely often disabled. Then by condition 4, R is never enabled after position i as if R was enabled after i, it would be kept enabled. Now by Condition 2 and Condition 3, either H or R is always enabled after position i. However, based on Definition 3.8 and weak fairness assumption on H, both H and R could be disabled at some point after i. And this can be used to derive a contradiction.

Given that each trigger event would be followed by the corresponding response event, Condition 1 is then used to avoid the recurring of trigger events between the trigger-response pair. □

3.3 Infinite Trace Refinement

Abstraction and refinement are usually essential to manage the complexity of modeling and reasoning about a system. Refinement of a system usually involves changing the variables of the system (Back, 1989). Data refinement is used to add more details to the data structure in the model, either by replacing existing variables or adding new variables to the model. In Event-B machines, gluing invariants are used to link the variables in the refined model to the variables in the abstract model.

We use $M =< S, S_0, E, K >$ to denote the abstract machine, which is data-refined to the concrete machine $M' =< S', S'_0, F, K' >$. Syntactically the abstract and concrete state spaces are represented by the possible values of the variables v and w respectively. S'_0 is defined by a predicate $L(w)$. The transition relation K' of an event $e \in F$ is defined by its guard $H_e(w)$ and action predicate $R_e(w)$. In general, gluing invariants define a relational mapping between concrete and abstract states. Instead of assuming a mapping function that maps states $s \in S'$ to states $s' \in S$, we assume $J \in S \leftrightarrow S'$ as a gluing relation that relates the states of M and M'. Syntactically, J is represented by a predicate $J(v, w)$. Event mapping function $g \in F \to E \cup \{skip\}$ is a total function from refined event labels to abstract event labels and *skip*. The *skip* events are mapped from concrete new events in M'. In the following section, we lift the event function to event traces in (3.1). The infinite trace model allows us to give a proper treatment of timed traces. In the following section we use event trace inclusion to define the refinement semantics between different machines.

$$g[v] = u \equiv \forall i \cdot i \in \mathrm{dom}(v) \Rightarrow u(i) = g(v(i)) \tag{3.1}$$

In refinements, additional guards could be imposed on events, which might cause deadlock that no events are enabled in the refined machine. To guarantee that the refined machine models the behavior of real-time systems, we use Lemma 3.15 to show that traces of the refined machine are infinite traces when M' is deadlock free relative to M.

Lemma 3.15. *Given that machine M' refines M and M is not deadlocked. Then all traces in $traces(M')$ are infinite traces when M' is deadlock free relative to M: $J[dom(K)] \subseteq dom(K')$*

Proof. Assume that there exists one trace $v \in traces(M')$ that is deadlocked at index l, formally presented as $v_s(l) \notin dom(K')$.

$$\begin{aligned}
& v_s(l) \notin dom(K') \\
\Rightarrow & \{J[dom(K)] \subseteq dom(K')\} \\
& v_s(l) \notin J[dom(K)] \\
\equiv & \{u_s(l) \mapsto v_s(l) \in J\} \\
& u_s(l) \notin dom(K)
\end{aligned}$$

As M is not deadlocked so $\forall i \cdot i \in \mathbb{N} \Rightarrow u_s(i) \in dom(K)$, this contradicts the result that $u_s(l) \notin dom(K)$. Thus all traces in $traces(M')$ are infinite traces. □

We first present the most straight-forward refinement semantics, which assumes that the concrete model does not introduce new events. Then we generalize the result to allow for new events. We treat traces for machines as compound traces of states and events. We first prove state trace refinement with forward simulation and Zorn's Lemma (Conrad, 2016), which then can be used to prove the event trace refinement.

3.3.1 Forward Simulation and Infinite State Trace Refinement

Forward simulation is a sufficient condition for refinement of systems with finite traces (Abrial, 2009). We show that forward simulation using a relational abstraction relation is sufficient for infinite trace refinement as well. Formally, forward simulation is defined in Definition 3.16. Figure 3.1 illustrates that the concrete machine M' simulates abstract M provided for each transition in M' that may lead from a set of states S'_i to a set of states S'_{i+1}, there exists a corresponding transition on the abstract machine M from an abstract state set S_i to a set of states S_{i+1}. The states S and S' are gluing related by J.

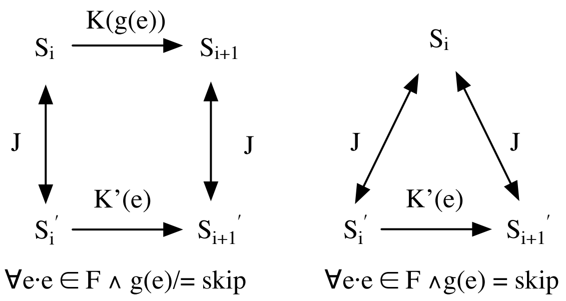

Figure 3.1: Forward Simulation

Definition 3.16 (Forward Simulation)**.** Let $M =< S, S_0, E, K >$ and machine $M' =< S', S'_0, F, K' >$. Let the event mapping function be $g \in F \to E \cup \{skip\}$. Let $J \in S \leftrightarrow S'$

be the gluing relation that relates the states of M and M', M is forward simulated by M' under K and K' provided:

$$\begin{cases} S_0' \subseteq J[S_0] \\ \forall e \cdot e \in F \wedge g(e) \neq \mathit{skip} \Rightarrow J; K'(e) \subseteq K(g(e)); J \\ \forall e \cdot e \in F \wedge g(e) = \mathit{skip} \Rightarrow J; K'(e) \subseteq J \end{cases}$$

Definition 3.17 (Relational Gluing Traces)**.** The infinite state trace u_s is defined to be gluing related to infinite state trace v_s as $J(u_s, v_s)$, formally:

$$J(u_s, v_s) \triangleq \forall i \cdot i \in \mathbb{N} \Rightarrow u_s(i) \mapsto v_s(i) \in J$$

In general, gluing invariants define a relational mapping between concrete and abstract states. We first define relational gluing traces in Definition 3.17 to link the states on the behavioral traces. $J(u_s, v_s)$ is used to relate the corresponding states in abstract and concrete traces. The notation $J[ts]$ is used to define the relational gluing trace image of state trace set ts, formally presented as (3.2).

$$J[ts] = \{v_s \mid u_s \in ts \wedge J(u_s, v_s)\} \tag{3.2}$$

Given that there are no stuttering events in the refinement step, then for any abstract event trace $u_e \in e\_traces(M)$, the corresponding concrete state trace set is the subset of the relational gluing image of the corresponding abstract state trace set. In other words, for each concrete state trace $v_s \in s\_traces(M', u_e)$, there exists an abstract state trace u_s that is gluing related with v_s and it is an abstract trace. For the case where there is no stuttering in the refinement step, Theorem 3.19 states that infinite state trace refinement could be proved with forward simulation. Based on the infinite state trace refinement, we provide the refinement semantics of event traces without new events in Definition 3.18. The behavior of the refined machine must be consistent with the behavior of the machine being refined during refinement (Robinson, 2012). Thus we use the event traces to define the refinement between machines. For machine M' to refine machine M, the concrete event trace set $e\_traces(M')$ must be the subset of the abstract event trace set $e\_traces(M)$ provided no new events in the refinement step.

Definition 3.18 (Event Trace Refinement Without New Events)**.** $M \sqsubseteq M'$ is defined as $e\_traces(M') \subseteq e\_traces(M)$.

Theorem 3.19. *Assume the event mapping between machine M' and M does not introduce new events. Then the state trace set of M' and some event trace v_e is a subset of the relational gluing image of the state trace set of M when M is forward simulated by M', formally:*

$$s\_traces(M', v_e) \subseteq J[s\_traces(M, g(v_e))] \tag{3.3}$$

Proof Outline.

$$\begin{aligned} & s\_traces(M', v_e) \subseteq J[s\_traces(M, v_e)] \\ \equiv & \forall v_s \in s\_traces(M', v_e) \Rightarrow \exists u_s \cdot J(u_s, v_s) \wedge u_s \in s\_traces(M, g(v_e)) \end{aligned}$$

To prove the above, we outline the proof steps as follows:

- We first construct a set U that contains the infinite abstract state traces or one of its prefixes related to each v_s by gluing invariant J in (3.4).
- Based on $S_0' \subseteq J[S_0] \wedge S_0' \neq \varnothing$ in Definition 3.16, $S_0 \neq \varnothing$. $s \in S_0$ is one of the elements in U. Thus the set U is not empty.
- Lemma 3.22 is constructed to show that infinite state traces in U are in $traces(M, g(v_e))$.
- We then show that any totally ordered set $Q \subseteq U$ has an upper bound in Lemma 3.24. That is, the main hypothesis of Zorn's lemma is satisfied.
- We use Zorn's Lemma to prove the existence of an infinite abstract state trace as a maximal element $u_{\mathcal{M}} \in U$ in a set of finite or infinite state traces that match the given infinite concrete trace in Lemma 3.25. Thus $u_{\mathcal{M}}$ is an infinite trace and $u_{\mathcal{M}} \in U$, which shows that $u_{\mathcal{M}}$ is an infinite abstract state trace. Then we use $u_{\mathcal{M}}$ as a witness for the existence of u_s to prove the theorem.

$$\begin{aligned} & U = \\ & \{u_s \mid u_s \in \mathbb{N} \to S \wedge \forall i \cdot i \geq 0 \Rightarrow u_s(0) \in S_0 \wedge u_s(i) \mapsto v_s(i) \in J \wedge u_s(i) \mapsto u_s(i+1) \\ & \in K(g(v_e(i)))\} \cup \{w_s \mid u_s \in 0..k \to S \wedge \exists k \forall i \cdot k > i \geq 0 \Rightarrow w_s \in (0..k-1) \to S \\ & \wedge w_s(0) \in S_0 \wedge w_s(i) \mapsto v_s(i) \in J \wedge w_s(i) \mapsto w_s(i+1) \in K(g(v_e(i)))\} \end{aligned} \tag{3.4}$$

We now proceed with the proof steps just outlined. In the first step, a set U that contains the infinite abstract traces and their prefixes related to v_s by gluing invariant J, formally presented as (3.4). We write $\preceq$ for the (standard) prefix order on U; that is, $u \preceq v$ when either u and v are both infinite and coincide, or u is a finite prefix of v. We define the prefix relation in Definition 3.20. Based on Lemma 3.21, $(U, \preceq)$ is a partial ordered set, which is antisymmetric, transitive, reflexive. Based on Definition 3.2, we show that the infinite traces in U are the state traces of the abstract machine M.

Definition 3.20 (Prefix Relation)**.** We define state trace u as a prefix v as $u \preceq v$, formally:

$$u \preceq v \triangleq u = (0..\#(u) - 1) \lhd v \tag{3.5}$$

Lemma 3.21. *$(U, \preceq)$ is a partial ordered set.*

Proof. We want to prove that $(U, \preceq)$ is reflexive, transitive and antisymmetric:

- $\forall u \cdot u \in U \Rightarrow u \preceq u$
- $\forall u, v, w \in U \wedge u \preceq v \wedge v \preceq w \Rightarrow u \preceq w$
- $\forall u, v \in U \wedge u \preceq v \wedge v \preceq u \Rightarrow u = v$

$$\begin{aligned} & u = (0..\#(u) - 1) \lhd u \\ \Rightarrow & u \preceq u \end{aligned}$$

$$\begin{aligned} & u \preceq v \wedge v \preceq w \\ \equiv & u = (0..\#(u) - 1) \lhd v \wedge v = (0..\#(v) - 1) \lhd w \\ \equiv & u = (0..\#(u) - 1) \lhd w \\ \equiv & u \preceq w \end{aligned}$$

$$\begin{aligned} & u \preceq v \wedge v \preceq u \\ \equiv & u = (0..\#(u) - 1) \lhd v \wedge v = (0..\#(v) - 1) \lhd u \\ \Rightarrow & u = v \end{aligned}$$

□

Lemma 3.22. *The infinite traces of U are in traces$(M, g(v_e))$.*

Proof. The infinite traces u_s satisfy the trace properties defined in Definition 3.2, thus any $u \in \{u_s \mid \forall i \cdot i \geq 0 \Rightarrow u_s(0) \in S_0 \wedge u_s(i) \mapsto v_s(i) \in J \wedge u_s(i) \mapsto u_s(i+1) \in K(g(v_e(i)))\}$ is in *traces*$(M, g(u_e))$. □

Lemma 3.24 states that any totally ordered chain $Q \subseteq U$ has an upper bound. There are two cases for the totally ordered set Q. One case is when Q contains only finite trace. In this case, the upper bound of Q are the traces of size i. The other case is when Q contains some infinite traces, that is, for each $i \in \mathbb{N}$, there exists a partial trace in the set Q that has size no less than i. To help prove the second case, Lemma 3.23 proves that if any two state traces in a totally ordered set Q are defined on index $i \in \mathbb{N}$, then the states at index i in the two traces are the same.

Lemma 3.23. *Given a totally ordered set $Q \subseteq U$, $\forall i, u, u' \cdot u \in Q \wedge u' \in Q \wedge i \in dom(u) \wedge i \in dom(u') \Rightarrow u(i) = u'(i)$.*

Proof.

$$\begin{aligned}
&\forall i, u, u' \cdot u \in Q \wedge u' \in Q \wedge i \in dom(u) \wedge i \in dom(u') \\
\equiv&\{\text{Q is totally ordered set}\} \\
&\{\forall u, u' \in Q \Rightarrow u \preceq u' \vee u' \preceq u\} \\
\Rightarrow&\begin{cases} u(i) = u'(i) & \text{where } u \preceq u' \wedge i \in dom(u) \\ u(i) = u'(i) & \text{where } u' \preceq u \wedge i \in dom(u') \end{cases} \\
\equiv& u(i) = u'(i)
\end{aligned}$$

□

Lemma 3.24. *Given U in (3.4), any totally ordered set $Q \subseteq U$ has an upper bound.*

Proof. Case 1: Q contains only finite trace. That is, all partial traces in the chain have size at most i, formally:

$$\exists i \in \mathbb{N} \Rightarrow (\forall u \in Q \Rightarrow \#(u) \leq i)$$

In this case, the upper bound of the chain is the partial trace whose size is with length i. As the set Q is totally ordered, so $\forall u, u' \in Q \Rightarrow u \preceq u' \vee u' \preceq u$. If the size of a trace is larger than i, then it contradicts the assumption that $\forall u \in Q \Rightarrow \#(u) \leq i$.

Case 2: Q contains some infinite traces. For each $i \in \mathbb{N}$, there exists a partial trace in the chain that has size no less than i, formally:

$$\forall i \cdot i \in \mathbb{N}, \exists u \in Q \wedge \#(u) \geq i \tag{3.6}$$

In this case we define a u_ω as $\forall i, u \cdot i \in \mathbb{N} \wedge \#(u) \geq i \Rightarrow u_\omega(i) = u(i)$. u_ω is well defined based on Lemma 3.23 as for all traces that are defined in index i, the state in index i is the same. First we prove that $u_\omega \in U$.

$$\begin{aligned}
&\forall i \cdot i \in \mathbb{N}, \exists u \in Q \wedge \#(u) \geq i \\
\Rightarrow&\{\text{Definition for } U \text{ in (3.4)}\} \\
&\forall i \cdot i \in \mathbb{N}, \exists u \wedge u(i) \mapsto v_s(i) \in J \wedge u(i) \mapsto u(i+1) \in K(g(v_e(i))) \\
\equiv&\{\text{let } u_\omega(i) = u(i) \text{ and } u_\omega(i+1) = u(i+1) \text{ where } \#(u) \geq i+1\} \\
&\forall i \cdot i \geq 0 \Rightarrow u_\omega(i) \mapsto v_s(i) \in J \wedge u_\omega(i) \mapsto u_\omega(i+1) \in K(g(v_e(i)))
\end{aligned}$$

Based on the Definition of u_ω, $\forall u \in Q \Rightarrow u \preceq u_\omega$. Thus u_ω is the upper bound for the totally order set Q. □

Given that the set U is a nonempty partially ordered set and every totally ordered set $Q \subseteq U$ has an upper bound, then there is a maximal element in U based on Zorn's

Lemma. Forward simulation is used to prove that the maximal element is an infinite trace in Lemma 3.25, which can be used to choose $u_s \in traces(M, u_e)$.

Lemma 3.25. *Assume M is forward simulated by M'. The set U in Formula (3.4) has a maximal element and that maximal element is an infinite trace.*

Proof. Based on Zorn's Lemma, then U has a maximal element $u_{\mathcal{M}} \in U$. Assume $u_{\mathcal{M}}$ is a finite trace with length k, then $u_{\mathcal{M}}(k-1) \notin dom(K(g(v_e(k-1))))$. Given the infinite state trace v_s and forward simulation, we can prove that $u_{\mathcal{M}}(k-1) \in dom(K(g(v_e(k-1))))$ with the following proof, which derives a contradiction. Thus $u_{\mathcal{M}}$ is an infinite trace.

$$\begin{aligned}
&\{u_{\mathcal{M}} \in U \text{ and } v_s \text{ is infinite}\}\\
\Rightarrow & u_{\mathcal{M}}(k-1) \mapsto v_s(k-1) \in J \wedge v_s(k-1) \mapsto v_s(k) \in K'(v_e(k-1))\\
&\{\text{Definition 3.16}\}\\
\Rightarrow & \exists s \cdot s \mapsto v_s(k) \in J \wedge u_{\mathcal{M}}(k-1) \mapsto s \in K(g(v_e(k-1)))\\
\equiv & u_{\mathcal{M}}(k-1) \in dom(K(g(v_e(k-1))))
\end{aligned}$$

□

Proof of Lemma 3.25 means that the proof of Theorem 3.19 is now complete. Then we use Theorem 3.26 to prove that forward simulation is enough to prove a valid refinement between abstract and concrete machines.

Theorem 3.26. *$M \sqsubseteq M'$ given M' is forward simulated by M and there are no new events in the refinement step.*

Proof.

$$\begin{aligned}
&\forall v \cdot v \in e\_traces(M')\\
\equiv & \{\text{Fix } v \in e\_traces(M') \text{and Lemma 3.5}\}\\
& s\_traces(M', v) \neq \varnothing\\
\Rightarrow & \{\text{Infinite State Trace Refinement: Theorem 3.19}\}\\
& s\_traces(M, g(v)) \neq \varnothing\\
\Rightarrow & \{\text{Lemma 3.5 and } g(v) = u\}\\
& u \in e\_traces(M)
\end{aligned}$$

□

3.3.2 Hiding Operator

In the previous section, we assumed that there are no new events introduced in the refined machine. In practice, adding new events in the refined model would make the behavior steps more fine grained. Figure 3.2 shows two cases where the concrete trace

$$\begin{tikzcd}
\mathrm{u\_s(0)} \arrow[r, "\mathrm{u\_e(0)}"] \arrow[d, leftrightarrow, "J"'] & \mathrm{u\_s(1)} \arrow[r, "\mathrm{u\_e(1)}"] \arrow[d, leftrightarrow, "J"'] & \mathrm{u\_s(2)} \arrow[r, "\mathrm{u\_e(2)}"] \arrow[d, leftrightarrow, "J"'] & {} & \cdots & \mathrm{u\_s(n)} \arrow[r, "\mathrm{u\_e(n)}"] \arrow[d, leftrightarrow, "J"'] & \mathrm{u\_s(n+1)} & \cdots \\
\mathrm{v\_s(0)} \arrow[r, "\mathrm{v\_e(0)}"'] & \mathrm{v\_s(1)} \arrow[r, "\mathrm{v\_e(1)}"'] & \mathrm{v\_s(2)} \arrow[r, "\mathrm{v\_e(2)}"'] & {} & \cdots & \mathrm{v\_s(n)} & &
\end{tikzcd}$$

(A) Deadlock in Concrete Trace

$$\begin{tikzcd}
\mathrm{u\_s(0)} \arrow[r, "\mathrm{u\_e(0)}"] \arrow[d, leftrightarrow, "J"'] & \mathrm{u\_s(1)} \arrow[r, "\mathrm{u\_e(1)}"] \arrow[d, leftrightarrow, "J"'] & \mathrm{u\_s(2)} \arrow[r, "\mathrm{u\_e(2)}"] \arrow[d, leftrightarrow, "J"'] & {} & \cdots & \mathrm{u\_s(n)} \arrow[loop above, "\mathrm{skip}"] \arrow[r, "\mathrm{u\_e(n)}"] \arrow[d, leftrightarrow, "J"'] \arrow[dr, leftrightarrow, "J"] \arrow[drr, leftrightarrow, "J"] & \mathrm{u\_s(n+1)} & \cdots \\
\mathrm{v\_s(0)} \arrow[r, "\mathrm{v\_e(0)}"'] & \mathrm{v\_s(1)} \arrow[r, "\mathrm{v\_e(1)}"'] & \mathrm{v\_s(2)} \arrow[r, "\mathrm{v\_e(2)}"'] & {} & \cdots & \mathrm{v\_s(n)} \arrow[r, "\mathrm{v\_e(n)}"'] & \mathrm{v\_s(n)} \arrow[r] & \cdots
\end{tikzcd}$$

(B) Infinite New Events in Concrete Trace

FIGURE 3.2: State Trace Refinement with Deadlock and Infinite New Events

does not simulate the abstract trace. In the first case, the concrete trace comes to a deadlock. In our definition of timed systems, time should always progress. So the traces of the refined model should also be infinite traces. Based on our setting, additional conditions are required to exclude the behaviors that would cause deadlocks in the refined machine. In the second case, the introduced new events occur infinitely often from some point n in the concrete trace, which can only be mapped to the prefix of the abstract trace. Thus we want to show that the event traces that hide the stuttering *skip* events do correspond to abstract traces. Jifeng (1989b) defines a hiding operator on labeled transition systems and develops simulations that allow the concrete model to have hidden transitions. Butler shows that in CSP, hiding an infinite behavior causes the process to diverge (Butler, 1992). Inspired by their work, we define a hiding operator on infinite event traces by using a hiding function h_D on an infinite set $D \subseteq \mathbb{N}$ in Definition 3.27. The hiding function h_D defines a bijection function from natural numbers to the infinite subset D. We define the function recursively. We map $h_D(0)$ to the minimal number in D since D is well-ordered. And $h_D(n+1)$ is mapped to the minimal number in the set $\{x \mid x \in D \wedge x > h_D(n)\}$. If D is finite, then h_D is not well-defined.

Definition 3.27 (Hiding Function)**.** Given an infinite set $D \subseteq \mathbb{N}$, the hiding function $h_D \in \mathbb{N} \rightarrowtail\!\!\!\!\twoheadrightarrow D$ is formally defined as:

$$\begin{cases} h_D(n) = min(D), & n = 0 \\ h_D(n+1) = min(\{x \mid x \in D \wedge x > h_D(n)\}), & \forall n \cdot n \in \mathbb{N}_1 \end{cases}$$

Firstly we want to prove that the hiding function is order isomorphic. Based on the order isomorphic function Definition 3.28 defined in (Devlin, 1979), it is clear that the hiding function is order isomorphic when D is infinite. Here well-ordered sets are totally ordered sets whose non-empty subsets have a least element in the ordering (Devlin, 1979). Since D and $\mathbb{N}$ are well-ordered sets, then h_D is unique based on the unique order isomorphism theorem between well-ordered sets provided in Theorem 3.29.

Definition 3.28 (Order Isomorphic Function (Devlin, 1979))**.** Let P, Q be well-ordered sets. A function $f \in P \rightarrowtail\!\!\!\!\twoheadrightarrow Q$ is defined as *order isomorphism* as $P \cong Q$ **iff**

$$P \cong Q \triangleq \forall x, y \cdot x \in P \land y \in P \land x < y \Rightarrow f(x) < f(y)$$

Theorem 3.29 (Unique Order Isomorphism between Well-ordered Sets (Devlin, 1979))**.** *Let P, Q be well-ordered sets. If $P \cong Q$, there is exactly one order-isomorphism $f \in P \rightarrowtail\!\!\!\!\twoheadrightarrow Q$.*

Then we define the hiding operator in infinite event traces in Definition 3.30 by using the hiding function to define the event trace $u_e \setminus A$ with all events in A removed in u_e. The hiding operation is the backward composition of a range subtraction and the hiding function. The range subtraction removes the indexes together with the events in A. The hiding function h_D remaps the natural numbers to the remaining events. We use Lemma 3.31 to show that $v_e \setminus A$ is a well defined infinite trace provided that v_e does not end with an infinite suffix of events in A.

Definition 3.30 (Hiding Operator for Infinite Event Traces)**.** Given an event trace $u_e \in \mathbb{N} \to E$ and a set of events A. Let $D = dom(u_e \rhd A)$, then the event trace $u_e \setminus A$ is formally defined as:

$$u_e \setminus A \triangleq (u_e \rhd A) \circ h_D \tag{3.7}$$

Lemma 3.31. *Given the infinite event trace $v_e \in \mathbb{N} \to E$, if v_e has an infinite suffix of events in A, then $v_e \setminus A$ is a finite trace.*

Proof. Given the event trace v_e with an infinite suffix of events in A, formally:$\exists j \in \mathbb{N} \Rightarrow \forall i \cdot i > j \land v_e(i) \in A$. Then it is clear that $\forall k \cdot k \in dom(v_e \rhd A) \Rightarrow k \leq j$. Let $h_D(n) = j$, then $\{x \mid x \in dom(v_e \rhd A) \land x > h_D(n)\} = \varnothing$. So $h_D(n+1)$ is not well-defined. Assume $v_e \setminus A$ is an infinite trace, then h_D is well defined for $\forall n \cdot n \in \mathbb{N}_1$, which derives a contradiction. So $v_e \setminus A$ is a finite trace. □

3.3.3 Generic Refinement Semantics

In Section 3.3, we assume that the concrete model does not introduce new events. In this section, we extend the refinement semantics 3.18 with the more generic Definition 3.32 by using the event mapping function $g \in F \to E \cup \{skip\}$. In the extended definition, we show that $M \sqsubseteq M'$ if for any concrete event trace $v_e \in e\_traces(M')$, there exists an abstract trace $u_e \in e\_traces(M)$, and $g(v_e) \setminus skip = u_e$.

Definition 3.32 (Generic Event Trace Refinement)**.**

$$M \sqsubseteq M' \equiv \forall v_e \in e\_traces(M') \Rightarrow (g(v_e) \setminus \{skip\}) \in e\_traces(M)$$

In timed systems, the behavioral traces are infinite and time should always progress. There are two properties to be preserved during the refinement step for timed models. Firstly, the refined behavioral trace should be infinite, provided the abstract trace is infinite. Secondly, introducing new events in the refinement step should not lead to divergence. Besides forward simulation, additional conditions are required to preserve these two properties. To simplify the proof, we construct intermediate traces $traces(M^*)$ with stuttering events *skip* in Definition 3.33. The property that all traces in M' are convergent with respect to new events requires that an intermediate event trace $\hat{v}_e \in e\_traces(M^*)$ does not have an infinite suffix of *skip* events under the condition that the abstract machine M is not deadlocked. Based on Lemma 3.31, if $\hat{v}_e$ has an infinite suffix of *skip* events, then $\hat{v}_e \setminus skip$ is a finite trace, which cannot be a trace of an abstract machine that is not deadlocked.

Definition 3.33 (Intermediate Traces with Stuttering Events)**.** The set of infinite traces $traces(M^*)$ of a machine M^* that includes stuttering events is defined as:

$$\begin{aligned} traces(M^*) \triangleq \{ & (\hat{v}_s, \hat{v}_e) \mid \hat{v}_s \in \mathbb{N} \rightarrow S \wedge \hat{v}_e \in \mathbb{N} \rightarrow (E \cup \{skip\}) \\ & \wedge \hat{v}_s(0) \in init \\ & \wedge \forall i \cdot i \geq 0 \wedge \hat{v}_e(i) \neq skip \Rightarrow \hat{v}_s(i) \mapsto \hat{v}_s(i+1) \in K(\hat{v}_e(i)) \\ & \wedge \forall i \cdot i \geq 0 \wedge \hat{v}_e(i) = skip \Rightarrow \hat{v}_s(i) = \hat{v}_s(i+1)\} \end{aligned}$$

The intermediate event trace maps concrete event labels F to abstract labels E and *skip*. We use Lemma 3.34 to show that the concrete event trace set conforms to the behavior of intermediate event trace set if M' is forward simulated by M.

Lemma 3.34. *Given machine M and M' with relabelling function g that allows new events, then $g[e\_traces(M')] \subseteq e\_traces(M^*)$ provided M is forward simulated by M'.*

Proof.

$$
\begin{aligned}
& \forall v_e \in e\_traces(M') \\
\Rightarrow & \{\text{Fix } v_e \in e\_traces(M') \text{ and Lemma 3.5}\} \\
& s\_traces(M', v_e) \neq \varnothing \\
\Rightarrow & \{\text{Infinite State Trace Refinement: Theorem 3.19}\} \\
& s\_traces(M^*, g(v_e)) \neq \varnothing \\
\Rightarrow & \{\text{Lemma 3.5 and } u_e = g(v_e)\} \\
& u\_e \in e\_traces(M^*)
\end{aligned}
$$

□

Theorem 3.35 is used to show additional rules required to prove $M \sqsubseteq M'$ and the traces in M' are convergent with respect to new events $\mathcal{N}$. Forward simulation is used to prove trace inclusion of concrete event traces and the intermediate event traces with stuttering events. $\hat{v}_e \setminus skip$ is one of the abstract events only if it is an infinite trace. Based on Lemma 3.31, $\hat{v}_e \setminus skip$ is an infinite trace if $\hat{v}_e$ does not have an infinite suffix of *skip* events. Conditional convergence and weak fairness conditions are required to prove the infiniteness of the behavioral trace.

Theorem 3.35. *Given M with transition relation K and M' with transition relation K'. Let F be the set of event labels in M and $\mathcal{N}$ be the introduced new events in M'. $M \sqsubseteq M'$ provided the following conditions hold:*

- *M forward simulated by M'.*
- *M' is deadlock free relative to M: $J[dom(K)] \subseteq dom(K')$.*
- *M' is weakly $(F \setminus \mathcal{N})$-fair.*
- *Events $\mathcal{N}$ in machine M' are conditional convergent under the condition that events $F \setminus \mathcal{N}$ are disabled;*

Proof Outline. We outline the proof with the following steps:

- We first prove that $g[e\_traces(M')] \subseteq e\_traces(M^*)$ provided M' is forward simulated by M with Lemma 3.34. Then we prove $\forall \hat{v}_e \cdot \hat{v}_e \in e\_traces(M^*) \Rightarrow \hat{v}_e \setminus skip \in e\_traces(M)$.
- $traces(M')$ are infinite traces based on Lemma 3.15.
- We then prove by contradiction that the intermediate event trace $\hat{v}_e \in e\_traces(M^*)$ does not end with an infinite suffix of skip events. Assume that $\hat{v}_e$ ends with an infinite suffix of skip events. As $g(v_e) = \hat{v}_e$, then the concrete trace v_e ends with

an infinite suffix of new events $\mathcal{N}$. In the case that $(F \setminus \mathcal{N})$ is disabled infinite many times in the suffix, based on the conditional convergence property, the new events will eventually be disabled and there cannot be an infinite suffix of new events. In the case that $(F \setminus \mathcal{N})$ is continuously enabled in the suffix, based on the weak fairness assumption some event $e \in (F \setminus \mathcal{N})$ will eventually occur. Thus $\hat{v}_e$ does not end with an infinite suffix of skip events. Based on Lemma 3.31, the intermediate event trace $\hat{v}_e \setminus skip$ is one of the abstract traces.

New variables and events are introduced to Event-B models in data refinements. Based on the invariants $I(v)$ of machine M and gluing invariants $J(v, w)$ that relate the abstract and concrete variables between M and N, Abrial (2009) demonstrates how the translation of the forward simulation rule yields to various proof obligations, such as invariant preservation (INV), feasibility (FIS), guard strengthening (GRD) and so on, that are used by the Rodin (Jastram and Butler, 2014) platform. In this section, we explore the refinement rules in terms of timed systems with infinite traces and infinitely many *Tick* events. Besides the forward simulation rule, the relative deadlock freedom rule, the conditional convergence rule, and weak fairness assumptions are required to prove that the refinement of a timed system still preserves the property that time can always proceed. The relative deadlock freeness rule guarantees that the refinement of a machine with infinite behavioral traces is always a machine with infinite traces. Conditional convergence and weak fairness assumptions are required to prevent new events from keeping occurring while the global clock can never proceed.

3.4 Conclusion

To summarize, our work with refinement semantics and trace semantics of timing properties has yielded two main contributions. First, we provided the trace semantics for trigger-response properties and bounded timing properties. Sufficient conditions were provided for the traces of an Event-B machine to satisfy these properties. Secondly, additional conditions were provided to verify refinement in Event-B models in terms of infinite behavioral traces. Zorn's Lemma was used together with forward simulation to prove infinite state/event trace refinement in refinement steps.

Chapter 4

Refinement of Timing Properties in Event-B Models

Based on the trace semantics that refines Event-B models with real-time trigger-response properties in Chapter 3, we develop refinement semantics for real-time properties in this chapter. We first present the refinement semantics of real-time properties. Formal proofs are presented with the hiding operator to refine the abstract timing properties into sequential or alternative sub-timing properties. Then a two-step refinement strategy is developed to refine the timing properties.

4.1 Refinement Semantics of Real-time Properties

Based on Definition 3.13, machines with timing properties could be described with infinite traces. In this section, we use Definition 4.1 to define the refinement semantics of machines with real-time properties. The timed behavior of a concrete timed model should also be consistent with the timed behavior in the abstract model. In our refinement strategies for timed systems with Event-B models, the abstract machine is extended with real-time trigger-response properties with no intermediate events between trigger and response events. Intermediate events are introduced in refinement steps.

Definition 4.1 (Refinement Semantics of Machines with Real-time Properties)**.** Refinement between machines with real-time properties in (4.1) is defined as (4.2).

$$M \wedge \mathit{Resp}(T, R, w, d) \sqsubseteq_g M' \wedge \mathit{Resp}(T', R', w', d') \tag{4.1}$$

$$\begin{aligned} &\forall v_e \cdot v_e \in e\_traces(M') \wedge v_e \models \mathit{Resp}(T', R', w', d') \\ &\Rightarrow \exists u_e \in e\_traces(M) \wedge u_e = (g(v_e) \setminus \mathit{skip}) \wedge u_e \models \mathit{Resp}(T, R, w, d) \end{aligned} \tag{4.2}$$

To account for the infinite behavior of timed traces, we use Lemma 4.2 to show convergence conditions and fairness assumptions under which $M \sqsubseteq M'$. Based on $M \sqsubseteq M'$, we also prove that the traces of the refined machine with intermediate events satisfy the trigger-response property in Lemma 4.2. Lemma 4.3 is used to show that given $M \sqsubseteq M'$, the timed property (that time should always progress) is preserved in the refinement step.

Lemma 4.2. *Let M be an Event-B machine with trigger-response pair (T, R) and R is enabled immediately after the execution of an event in T. Let $g[T'] = T$ and $g[R'] = R$. We define $H' \subseteq F$ as intermediate events between T' and R' in M' and $g[H'] = \{skip\}$. Assume M is weakly fair with respect to R and M' is weakly fair with respect to $H' \cup R'$. If M' simulates M under J, H' is convergent under the condition of R' is disabled and M' is relative deadlock free to M, then $M \sqsubseteq M'$ and M'* ***satisfies*** *$TR(T', R')$.*

Proof. H' are new events introduced to M'. Based on Theorem 3.35, $M \sqsubseteq M'$.

$$
\begin{aligned}
&\{M \sqsubseteq M'\}\\
\equiv&\forall v_e \cdot v_e \in e\_traces(M') \Rightarrow \exists u_e \cdot u_e \in e\_traces(M) \wedge u_e = g(v_e \setminus \mathcal{H}') \wedge u_e \models TR(T, R)\\
\equiv&\{\text{ Fix } v_e \in e\_traces(M') \text{ and Definition 3.9}\}\\
&\forall i \cdot i \geq 0 \wedge u_e(i) \in T \Rightarrow (\exists j \cdot j > i \wedge u_e(j) \in R \wedge \forall k \cdot i < k < j \Rightarrow u_e(k) \notin T)\\
\Rightarrow&\{u_e = g(v_e \setminus \mathcal{H}') \text{ , let } D = dom(v_e \rhd H')\}\\
&\forall i \cdot i \geq 0 \wedge u_e(i) \in T \wedge v_e(h_D(i)) \in g^{-1}[T] \Rightarrow (\exists j \cdot j > i \wedge u_e(j) \in R \wedge u_e(h_D(j)) \in g^{-1}[R]\\
&\wedge \forall k \cdot h_D(i) < k < h_D(j) \Rightarrow u_e(k) \notin g^{-1}[T])\\
\equiv&v_e \models TR(T', R')
\end{aligned}
$$

□

Lemma 4.3. *Given $M \sqsubseteq M' \wedge g^{-1}[Tick] = Tick$ and $\forall u_e \cdot u_e \in e\_traces(M) \Rightarrow timed(u_e)$, then $\forall v_e \cdot v_e \in e\_traces(M') \Rightarrow timed(v_e)$.*

Proof.

$$
\begin{aligned}
&\{M \sqsubseteq M'\}\\
\equiv&\forall v_e \cdot v_e \in e\_traces(M') \Rightarrow \exists u_e \cdot u_e \in e\_traces(M) \wedge u_e = g(v_e \setminus \mathcal{H}') \wedge timed(u_e)\\
\equiv&\{\text{ Fix } v_e \in e\_traces(M') \text{ and Definition 3.11}\}\\
&\forall i \cdot i \in \mathbb{N} \Rightarrow (\exists j \cdot j \geq i \wedge u_e(j) = g^{-1}[Tick])\\
\Rightarrow&\{u_e = g(v_e \setminus \mathcal{H}') \text{ , let } D = dom(v_e \rhd H')\}\\
&\forall i \cdot i \in \mathbb{N} \Rightarrow (\exists j \cdot j \geq i \wedge u_e(j) = Tick \wedge v_e(h_D(j)) = Tick)\\
\equiv&timed(v_e)
\end{aligned}
$$

□

Lemma 4.4 proves that all traces of M' that satisfy the timing property $Resp(T', R', w', d')$ correspond to traces of M that satisfy the original timing property $Resp(g[T'], g[R'], w, d)$ when $M \sqsubseteq M'$.

Lemma 4.4. *Given $M \sqsubseteq M'$, then $M \wedge Resp(g[T'], g[R'], w, d) \sqsubseteq_g M' \wedge Resp(T', R', w', d')$ if $w \leq w' \wedge d' \leq d$.*

Proof.

$$
\begin{aligned}
&\forall v_e \cdot v_e \in e\_traces(M') \wedge v_e \models Resp(T', R', w', d')\\
\equiv&\{\text{Fix } v_e \in e\_traces(M') \wedge v_e \models Resp(T', R', w', d')\}\\
&\forall i \cdot i \geq 0 \wedge v_e(i) \in T' \Rightarrow\\
&\exists j \cdot j > i \wedge v_e(j) \in R' \wedge (w' \leq \#(v_e[i,j] \upharpoonright Tick) \leq d')\\
\Rightarrow&\{M \sqsubseteq M' \wedge \forall v_e \cdot v_e \in e\_traces(M')\}\\
&\exists u_e \cdot u_e \in e\_traces(M) \wedge u_e = g(v_e \setminus \mathcal{H}')\\
\Rightarrow&\{\forall v_e \cdot v_e \in e\_traces(M') \wedge v_e \models Resp(T', R', w', d') \wedge \exists u_e = g(v_e \setminus \mathcal{H}')\\
&\text{, let } D = dom(v_e \rhd H')\}\\
&\forall i \cdot i \geq 0 \wedge v_e(i) \in T' \wedge u_e(h_D^{-1}(i)) \in g[T'] \Rightarrow\\
&\exists j \cdot j > i \wedge v_e(j) \in R' \wedge u_e(h_D^{-1}(j)) \in g[R'] \wedge (w' \leq \#(u_e[h_D^{-1}(i), h_D^{-1}(j)] \upharpoonright Tick) \leq d')\\
\Rightarrow&\{w \leq w' \wedge d' \leq d\}\\
&\forall i \cdot i \geq 0 \wedge v_e(i) \in T' \wedge u_e(h_D^{-1}(i)) \in g[T'] \Rightarrow\\
&\exists j \cdot j > i \wedge v_e(j) \in R' \wedge u_e(h_D^{-1}(j)) \in g[R'] \wedge (w \leq \#(u_e[h_D^{-1}(i), h_D^{-1}(j)] \upharpoonright Tick) \leq d)\\
\equiv&\{\text{Lemma 4.3 and Definition 3.12}\}\\
&u_e \models Resp(g[T'], g[R'], w, d)
\end{aligned}
$$

□

4.2 Refining Real-Time Trigger-Response Properties to Sequential Sub-Timing Properties

Consider an abstract model M of a system with real-time trigger-response property $Resp(T, R, w, d)$ and intermediate events with the event sequence that follows the order $T \rightarrow I \rightarrow R$. We refine the abstract timing property into two sequential sub-timing properties with $Resp(T, I, w_1, d_1) \wedge Resp(I, R, w_2, d_2)$. Lemma 4.5 proves that the timing property (T, R, w, d) is sequentially refined by a pair of timing properties $Resp(T, I, w_1, d_1)$ and $Resp(I, R, w_2, d_2)$.

Lemma 4.5. *Given $M \sqsubseteq M'$, then (4.3) holds if $w \leq (w_1 + w_2)$ and $(d_1 + d_2) \leq d$.*

$$M \wedge Resp(g[T'], g[R'], w, d) \sqsubseteq_g M' \wedge Resp(T', I, w_1, d_1) \wedge Resp(I, R', w_2, d_2) \qquad (4.3)$$

Proof.

$$
\begin{aligned}
&\forall v_e \cdot v_e \in e\_traces(M') \wedge v_e \models Resp(T', I, w_1, d_1) \wedge Resp(I, R', w_2, d_2)\\
\equiv&\{\text{Fix } v_e \in e\_traces(M') \wedge v_e \models Resp(T', I, w_1, d_1) \wedge Resp(I, R', w_2, d_2)\}\\
&\forall i \cdot i \geq 0 \wedge v_e(i) \in T' \Rightarrow\\
&\exists j \cdot j > i \wedge v_e(j) \in I \wedge (w_1 \leq \#(v_e[i,j] \upharpoonright Tick) \leq d_1)\\
&\wedge \exists k \cdot k > j \wedge v_e(k) \in R' \wedge (w_2 \leq \#(v_e[j,k] \upharpoonright Tick) \leq d_2)\\
\Rightarrow&\{M \sqsubseteq M' \wedge \forall v_e \cdot v_e \in e\_traces(M')\}\\
&\exists u_e \cdot u_e \in e\_traces(M) \wedge u_e = g(v_e \setminus \mathcal{H}')\\
\Rightarrow&\{\forall v_e \cdot v_e \in e\_traces(M') \wedge v_e \models Resp(T', I, w_1, d_1) \wedge Resp(I, R', w_2, d_2) \wedge u_e = g(v_e \setminus \mathcal{H}')\\
&\text{, let } D = dom(v_e \rhd H')\}\\
&\forall i \cdot i \geq 0 \wedge v_e(i) \in T' \wedge u_e(h_D^{-1}(i)) \in g[T'] \Rightarrow\\
&\exists k \cdot k > j \wedge v_e(k) \in R' \wedge u_e(h_D^{-1}(k)) \in g[R]\\
&\wedge (w_1 + w_2 \leq \#(v_e[h_D^{-1}(i), h_D^{-1}(k)] \upharpoonright Tick) \leq d_1 + d_2)\\
\Rightarrow&\{w \leq (w_1 + w_2) \text{ and } (d_1 + d_2) \leq d\}\\
&\forall i \cdot i \geq 0 \wedge v_e(i) \in T' \wedge u_e(h_D^{-1}(i)) \in g[T'] \Rightarrow\\
&\exists k \cdot k > i \wedge v_e(k) \in R' \wedge u_e(h_D^{-1}(k)) \in g[R'] \wedge (w \leq \#(u_e[h_D^{-1}(i), h_D^{-1}(k)] \upharpoonright Tick) \leq d)\\
\equiv&\{\text{Lemma 4.3 and Definition 3.12}\}\\
&u_e \models Resp(g[T'], g[R'], w, d)
\end{aligned}
$$

□

4.3 Refining Alternative Timing Properties to Sub-Timing Properties

In timed systems, there are cases that the response events are two alternative events R_1 and R_2, where R_1 and R_2 presents different scenarios of responses with the same timing property, presented as $M \wedge Resp(T, R_1 \cup R_2, w, d)$ where $R1 \cap R2 = \varnothing$. To refine the high level alternative response events to concrete alternative response events, alternative intermediate events are required to resolve the nondeterminism of alternative responses. Lemma 4.6 proves that the alternative timing property $Resp(T, R_1 \cup R_2, w, d)$ is refined by $Resp(T, I_1 \cup I_2, w_1, d_1)$, $Resp(I_1, R_1, w_2, d_2)$ and $Resp(I_2, R_2, w_3, d_3)$.

Lemma 4.6. *Given $M \sqsubseteq M'$, then (4.4) holds if $w \leq (w_1 + w_2) \wedge w \leq (w_1 + w_3)$ and $(d_1 + d_2) \leq d \wedge (d_1 + d_3) \leq d$.*

$$
\begin{aligned}
&M \wedge Resp(g[T'], g[R'_1] \cup g[R'_2], w, d) \sqsubseteq_g\\
&M' \wedge Resp(T', I_1 \cup I_2, w_1, d_1) \wedge Resp(I_1, R'_1, w_2, d_2) \wedge Resp(I_2, R'_2, w_3, d_3)
\end{aligned}
\tag{4.4}
$$

Proof.

$$
\begin{aligned}
& M' \land Resp(T', I_1 \cup I_2, w_1, d_1) \land Resp(I_1, R_1', w_2, d_2) \land Resp(I_2, R_2', w_3, d_3) \\
\equiv & (M' \land Resp(T', I_1, w_1, d_1) \land Resp(I_1, R_1', w_2, d_2)) \lor \\
& (M' \land Resp(T', I_r, w_1, d_1) \land Resp(I_2, R_2', w_3, d_3)) \\
\Rightarrow & \{Lemma\ 4.5\} \\
& M \land Resp(g[T'], g[R_1'] \cup g[R_2'], w, d) \lor M \land Resp(g[T'], g[R_1'] \cup g[R_2'], w, d)
\end{aligned}
$$

□

4.4 Two-step Refinement Strategy

Event-B has a strong and flexible refinement strategy described in (Hallerstede et al., 2013; Schneider et al., 2012). Based on the monotonicity of timing properties in Event-B models, we design the two-step refinement strategy to refine timed systems. In the first step, we introduce intermediate events to the abstract model while preserving the abstract timing properties with a strategy that has the following restrictions on the machines in the refinement chain $M_0 \sqsubseteq M_1 \sqsubseteq ... \sqsubseteq M_n$:

1. Each event of M_i is refined by at least one event of M_{i+1};
2. Given M_i, intermediate events are introduced as new events in M_{i+1};
3. M_{i+1} is deadlock free relative to M_i;
4. M_{i+1} is forward simulated by M_i;
5. The new events introduced in M_{i+1} are either anticipated or conditional convergent under the condition that the refined response events are disabled, M_{i+1} is weakly fair on these new events;
6. All anticipated events should be refined to conditional convergent events, no anticipated events remain in the final machine;
7. All refinement steps $M_i, 0 \leq i \leq n$ are weakly fair on the response events and *Tick* events;

In condition 5), new events introduced in the refined step are either *anticipated* or *conditional convergent* so that these events are never executed forever. A variant $V(v)$, that has to be decreased by every *convergent* event and must not be increased by *anticipated* events, needs to be introduced from new proof obligations to prevent the infinite execution of new events. The details that refine *anticipated* events to *conditional convergent* events could be deferred to later refinement steps. Nevertheless, these *anticipated* events should be eventually refined to *convergent* events in the refinement chain. To syntactically encode *conditional convergent* events in Event-B models, we have two versions of intermediate events included in the machine. The convergent intermediate event has the guards that indicate that the response events are disabled. The other intermediate event has the condition under which the response events are enabled is not marked as convergent. Based on Theorem 3.35 and Lemma 4.2, conditions 3)-7) can be used to guarantee that for all $i \in 0..n-1$, $M_i \sqsubseteq M_{i+1}$ and M_{i+1} satisfies the trigger-response property. Then in the second step, the abstract timing properties could be refined to sequential or alternative timing properties between trigger, intermediate and response events.

4.5 Conclusion

Based on the trace semantics and hiding operator defined in Chapter 3, we developed refinement rules to refine abstract timing properties into sequential or alternative subtiming properties. We also provided rules for a two-step refinement strategy to develop real-time systems in a stepwise manner. In Chapter 5, we used the bounded-retransmission protocol case study to illustrate the refinement conditions and fairness assumptions applied to the real-time systems.

Chapter 5

Bounded Re-transmission Protocol Case Study

In this chapter, we present the Bounded Retransmission Protocol (BRP) case study based on the additional proof obligations, weak fairness assumptions and refinement strategy proposed in Chapter 4. We first overview the BRP briefly. Then the two-step refinement strategy is used to model real-time properties in the BRP case study. We show how to encode the conditional convergent events in Event-B models. The relative deadlock freeness is encoded as additional invariants to ensure the refined machine satisfies the trigger-response properties. Finally, we compare our work with existing work and discuss the results.

5.1 Bounded Re-transmission Protocol

The BRP is a file transfer protocol that deals with the fault tolerance of the system under unreliable network communications (D'Argenio et al., 1997). The schematic view of the transmission protocol is shown in Figure 5.1. The transmitter sends the file to the receiver packet by packet through the data channel. As soon as the receiver receives the packet, it sends back an acknowledgment to the transmitter through an acknowledgment channel. When the transmitter receives the acknowledgment, it confirms that the packet is sent out successfully and sends out the next packet. As the data channel and the acknowledgment channel are not reliable, the packet might be lost, or the transmitter might not receive the acknowledgment. The transmitter will resend the packet to the receiver if it has not received confirmation of the same packet within some deadline. In the case of successive losses of messages, the process of packet re-transmission can be repeated several times with a retry counter. When the counter reaches a specific limit, the transmitter and the receiver decides to abort. The two-step refinement strategy could be used to resolve different levels of nondeterminism of intermediate events.

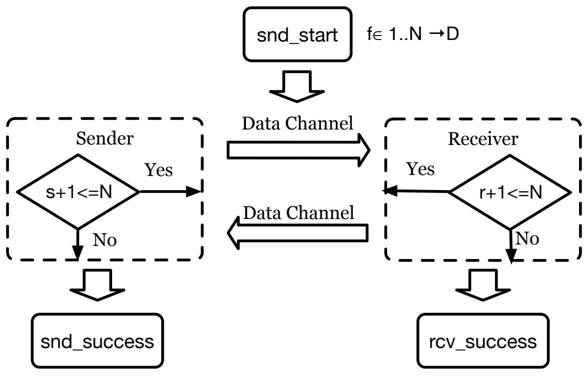

FIGURE 5.1: Scheme View of the Bounded Retransmission Protocol

Figure 5.2 shows the time diagram of BRP in different refinement levels based on the following timing requirements (TR).

TR-1 The transmitter should send the next packet within *pkt_ddl* once it receives the acknowledgment of the current packet.

TR-2 If the data channel is broken, the receiver should abort within *pkt_ddl* since the transmitter receives the acknowledgment of the last packet.

TR-3 If the data channel is working, then it takes at most *c_ddl* time units to transmit the packet from the transmitter to the receiver.

TR-4 If the data channel is broken, then it takes at most *pkt_abt_duration* time units for the receiver to abort since it receives the last packet.

TR-5 It takes the receiver at least $2 * c\_dly$ time units but at most *retry_ddl* time units before knowing that the transmitter has not sent the new packet.

TR-6 It takes the receiver at least $2 * c\_dly * retry\_num$ time units but at most $retry\_ddl * retry\_num$ time units to abort during which it has not received any packet.

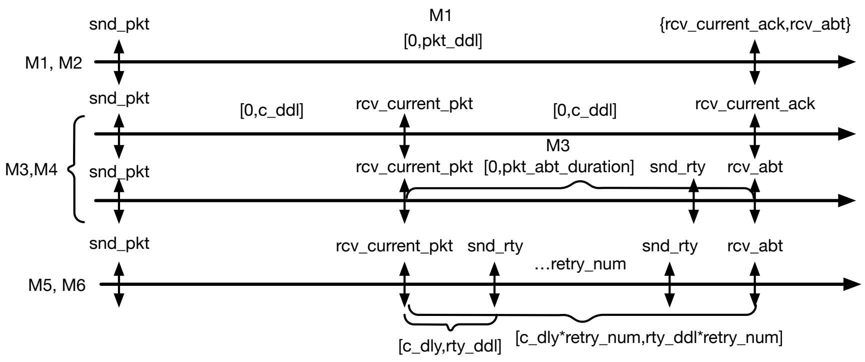

FIGURE 5.2: Time Diagram of BRP in Different Refinement Levels

Based on the timing requirements, we present our refinement strategy as follows to formalize BRP in Event-B models.

M_0 specifies the three main components of BRP, namely the transmitter, receiver and channel;

M_1 introduces timing properties to M_0 to model **TR-1** and **TR-2**;

M_2 introduces intermediate events to denote the packet could either be received by the receiver or resent by the transmitter;

M_3 introduces timing properties to M_2 to model **TR-3** and **TR-4**;

M_4 refines the *anticipated* re-send packet event to *conditional convergent* event;

M_5 introduces timing properties to M_4 to model **TR-5** and **TR-6**;

M_6 refines the *anticipated* receive packet event to *convergent* event;

5.2 Modeling the BRP

5.2.1 Abstract Machine

In the abstract machine we specify the two components in BRP with the machine given in Figure 5.3. The variable s presents the file pointer in the transmitter. The transmitter, receiver have three states, namely *working*, *success*, and *failure*. *Success* denotes the state that a file has been transmitted successfully, and the next file is ready to transmit. *Working* represents the state that the transmitter and receiver are transmitting the file packet by packet. *Failure* means that there is something wrong with the data channel or the acknowledgment channel, and the system has to abort. To start with the *snd_start* event, the transmitter state *s_st* and receiver state *r_st* are set to *working*. In the case the channel is working (*snd_pkt* event), the file pointer is increased to show that the transmitter transfers the last packet and receives the corresponding acknowledgment successfully. In the case that the packet or acknowledgment is lost during transmission, the receiver aborts the transmission with event *rcv_abt*. We use Property (5.1) to present the trigger-response property that once a packet is transmitted from the transmitter, either it is transmitted successfully, or the receiver abort. We use *snd_finish* and *rcv_finish* events to denote that the file is transmitted successfully for the transmitter and receiver respectively. *snd_abt* and *rcv_abt* events are used to denote that the transmitter and receiver abort when the channel is broken.

$$TR(snd\_pkt, \{snd\_pkt, rcv\_abt\}) \tag{5.1}$$

```
sets STATUS DATA

constants working success failure N file

axioms
  @axm0_1 partition(STATUS, {working}, {success}, {failure})
  @axm0_2 N∈ ℕ1
  @axm0_3 file∈ 1..N → DATA
end

invariants
@r_st_type r_st∈ STATUS
@s_st_type s_st∈ STATUS
@s_type s∈ 0..N
```

```
event snd_start
where
  @grd1 s_st=success
  @grd2 s=0
then
  @act1 s_st:= working
  @act2 r_st:= working
end

event snd_pkt
where
  @grd0_1 s_st=working
  @grd0_3 s+1≤ N
then
  @act0_1 s:= s+1
end

event snd_finish
where
  @grd2 s_st=working
  @grd0_4 s=N
then
  @act0_1 s_st:= success
end
```

```
event rcv_finish
where
  @grd1 r_st=working
  @grd2 s_st=success
then
  @act1 r_st:= success
end

event rcv_abt
where
  @grd1 r_st=working
then
    @act1 r_st:= failure
end

event snd_abt
where
  @grd1 s_st=working
then
    @act1 s_st:= failure
end
```

FIGURE 5.3: Machine $M0$ that Specifies the Abstract BRP Protocol

5.2.2 First Refinement

The trigger-response property (5.1) is then extended with timing properties specified in **TR-1** and **TR-2**. Figure 5.4 shows the extended Event-B model as the first refinement. A new *clock* variable is added to represent the current time. And we set the timestamps of *snd_pkt*, *rcv_abt* as *pkt_tp*, *pkt_tc* and *abt_t* respectively. *pkt_tp* denotes the timestamp of receiving the acknowledgment from the previous packet, *pkt_tc* presents the timestamp of current acknowledgment. @$inv1\_2$ and @$inv1\_4$ show the timing property (5.2). @$inv1\_1$ and @$inv1\_3$ serve as auxiliary invariants to support @$inv1\_2$ and @$inv1\_4$.

Some infinite event traces of machine $M1$ are shown in (5.3). Besides the invariants related to the time constraints between trigger and response events, we also show that the refined machine is relative deadlock-free to $M0$ with invariant $@dlf\_m1$. With all the proof obligations discharged, we can prove that $M0$ is forward simulated by $M1$. We also assume weak fairness on *snd_pkt*, *rcv_abt* and *tick* events collectively based on Lemma 4.2. In this refinement, we did not introduce intermediate events so that we do need to reason about conditional convergence on intermediate events. In later sections, we will gradually refine the machine with intermediate events with real-time properties. Also, refinement strategies and additional proof obligations will be presented.

$$Resp(snd\_pkt, \{snd\_pkt, rcv\_abt\}, 0, pkt\_ddl) \tag{5.2}$$

$$\begin{cases} < snd\_start, rcv\_abt, tick, tick, ... > \\ < snd\_start, snd\_pkt, rcv\_abt, tick, tick, ... > \\ < snd\_start, snd\_pkt, tick, rcv\_abt, tick, tick, ... > \\ < snd\_start, snd\_pkt, tick, snd\_pkt, tick, tick, ... > \end{cases} \tag{5.3}$$

```
@inv1_1 s_st=working  ⇒clk−pkt_tc≤  pkt_ddl
@inv1_2 pkt_tp≤  pkt_tc ⇒ pkt_tc−pkt_tp≤  pkt_ddl
@inv1_3 r_st=working  ⇒clk−pkt_tc≤  pkt_ddl
@inv1_4 pkt_tc≤ abt_t ⇒abt_t− pkt_tc≤  pkt_ddl
@dlf_m1 (s_st=success ∧ s=0) ∨ (s_st=working ∧ s+1≤ N) ∨ (s_st=working  ∧s=N)
∨ (r_st=working ∧ s_st=success) ∨ (r_st=working ) ∨ (s_st=working ) ∨ (s_st≠working  ∧ r_st≠
    working )
```

```
event snd_start extends snd_start
then
  @act1_1 pkt_tc:= clk
  @act1_2 pkt_tp:= clk
end

event snd_pkt extends snd_pkt
then
  @act1_1 pkt_tc:= clk
  @act1_2 pkt_tp:= pkt_tc
end
```

```
event rcv_abt extends rcv_abt
then
  @act1_1 abt_t:= clk
end

event tick
  where
    @grd1_1 s_st=working  ⇒clk+1−pkt_tc≤
      pkt_ddl
    @grd1_2 r_st=working  ⇒clk+1−pkt_tc≤
      pkt_ddl
  then
    @act1_1 clk:= clk+1
end
```

FIGURE 5.4: Machine $M1$ that Extends Trigger-Response Property (5.1) with Timing Properties Specified in **TR-1** and **TR-2**

5.2.3 Second Refinement

In the second refinement, we first introduce *rcv_current_pkt* and *snd_rty* as intermediate events to represent that the packet could either be received by the receiver or resent by the transmitter. The nondeterminism choice of intermediate events leads to alternative responses. The machine $M2$, given in Figure 5.5, introduces variables r and *rcv_file* to indicate that the transmitted file is denoted by *rcv_file* of length r in the receiver part. We use variable *c_st* to denote the status of the channel. In the case that the channel is in *working* state and the receiver receives the packet (*rcv_current_pkt* event), the file pointer r in the receiver is increased by one and the received file *rcv_file* receives the current packet successfully. In the case that the channel is in *failure* state then the transmitter re-sends the packet (*snd_rty* event). By using the two-step refinement strategy, we first use the Rodin tool to discharge the proof obligations to verify that $M2$ forward simulates $M1$. Invariant @*dlf_m2* is used to verify that $M2$ $M2$ is deadlock-free relative to $M1$. In the refinement, we want the *snd_rty* event to be conditional convergent when the channel is broken. Thus we encode the *snd_rty* event with one version with *anticipated* status and the other version *snd_rty_n* with *ordinary* status. The *anticipated* event *snd_rty* would be refined to *convergent* event in a future refinement step. Also, we set *rcv_current_pkt* as *anticipated* status so that it is refined to *convergent* later. Based on the conditions shown in Lemma 4.2, the behavioral traces of refined machine satisfy the specified real-time properties under weak fairness assumptions on *rcv_current_pkt*, *snd_rty*, *snd_pkt*, *rcv_abt* and *tick* events.

5.2.4 Third Refinement

In $M3$, we refine the abstract timing property (5.2) into three sub-timing properties shown in timing properties (5.5). Specifically, (5.2) is refined to (5.5a) and (5.5b) in the case that channel is working properly, and (5.5c) in the case that the channel is broken. We assume that the transmission delay and deadline for the channel are *c_dly* and *c_ddl*, respectively. Once the receiver received the packet, it should abort within *pkt_abt_duration* if the channel is broken. The relations between these timing properties are shown in (5.4).

$$\begin{cases} c\_ddl > c\_dly > 0 \\ pkt\_ddl > 2 * c\_ddl \\ pkt\_abt\_duration + c\_ddl < pkt\_ddl \end{cases} \tag{5.4}$$

As shown in Figure 5.6, we use @$grd3\_1$ and @$grd3\_2$ of *tick* event in $M3$ to replace @$grd1\_1$ of *tick* event in $M1$. @$inv3\_2$ and @$inv3\_4$ are used to show that (5.5a) and (5.5b) are satisfied in $M3$. @$inv3\_1$ and @$inv3\_3$ serve as auxiliary invariants to support @$inv3\_2$ and @@$inv3\_4$. In the case that the channel is broken, we use @$grd3\_3$ and @$grd3\_4$ of *tick* event in $M3$ to replace @$grd1\_2$ of *tick* event in $M1$. @$inv3\_6$ is

```
invariants
@c_st_type c_st∈ STATUS
@inv2_2 r∈ s..s+1
@inv2_3 rcv_file=0..r ◁ file
@s1 c_st=working⇒r_st=working
@dlf_m2 (s_st=success ∧ s=0) ∨ (s_st=working ∧ s+1≤ N ∧ r=s+1 ∧ c_st=working)
∨ (r_st=working ∧c_st=working ∧ s_st=working ∧ r<N ∧ r=s)
∨ (s_st=working ∧ r_st=working ∧ r=s+1) ∨ (s_st=working ∧ s=N ∧ s=r)
∨ (r_st=working ∧ s_st=success)∨ (s_st=working ∧ c_st=failure ∧ r=s+1)
∨ (r_st=working ∧ c_st=failure ∧ r=s+1)∨ (s_st≠working  ∧ r_st≠working )
```

```
event snd_start extends snd_start
then
  @act2_1 c_st:= working
end

event snd_pkt extends snd_pkt
  where
    @grd2_1 r=s+1
    @grd2_2 c_st=working
  then
    @act2_1 c_st:= working
end

event snd_finish extends snd_finish
where
  @grd2_1 s=r
end

event rcv_finish extends rcv_finish
then
  @act2 c_st:= success
end

event rcv_abt extends rcv_abt
where
@grd2_1 c_st=failure
@grd2_2 r=s+1
end

event snd_abt extends snd_abt
where
@grd2_1 c_st=failure
@grd2_2 r=s+1
end
```

```
anticipated event rcv_current_pkt
where
  @grd2_1 r_st=working
  @grd2_2 c_st=working
  @grd2_3 s_st=working
  @grd2_4 r<N
  @grd2_5 r=s
then
  @act2_1 r:= r+1
  @act2_2 rcv_file:= rcv_file ∪ {r+1 ↦  file(s
      +1)}
  @act2_3 c_st:∈ {working, failure}
end

anticipated event snd_rty
where
  @grd2 c_st=failure
  @grd3 s_st=working
  @grd4 r_st=working
  @grd5 r=s+1
then
  @act1 c_st:∈ {working, failure}
end

event snd_rty_n
where
  @grd2 c_st=working
  @grd3 s_st=working
  @grd4 r_st=working
  @grd5 r=s+1
then
  @act1 c_st:∈ {working, failure}
end
```

FIGURE 5.5: Machine $M2$ that introduces intermediate events to $M1$

used to present timing property (5.5c). To guarantee that $M3$ is relative deadlock free to $M2$, we construct invariant @dlf_m3 to verify that the refined machine would not be deadlocked.

$$Resp(snd\_pkt, rcv\_current\_pkt, 0, c\_ddl) \tag{5.5a}$$

$$Resp(rcv\_current\_pkt, snd\_pkt, 0, c\_ddl) \tag{5.5b}$$

$$Resp(rcv\_current\_pkt, rcv\_abt, 0, pkt\_abt\_duration) \tag{5.5c}$$

```
axioms
@axm3_1 pkt_ddl>2∗c_ddl
@axm3_2 pkt_abt_duration+c_ddl<pkt_ddl
invariants
@inv3_1 s_st=working ∧ r=s  ⇒clk−pkt_tc≤  c_ddl
@inv3_2 pkt_tc≤  pkt_rcv_t ⇒pkt_rcv_t − pkt_tc≤  c_ddl
@inv3_3 s_st=working ∧ r=s+1  ⇒clk−pkt_rcv_t≤  c_ddl
@inv3_4 pkt_rcv_t≤  pkt_tc ∧s≠0 ⇒pkt_tc − pkt_rcv_t≤  c_ddl
@inv3_5 r_st=working ∧ r=s+1 ⇒clk−pkt_rcv_t≤  pkt_abt_duration
@inv3_6 pkt_rcv_t≤ abt_t ⇒abt_t− pkt_rcv_t≤  pkt_abt_duration
@dlf_m3 (s_st=success ∧ s=0)∨ (s_st=working ∧ s+1≤ N ∧ r=s+1 ∧ c_st=working)
∨ (r_st=working ∧c_st=working ∧ s_st=working ∧ r<N ∧ r=s)
∨ (s_st=working ∧ r_st=working) ∨ (s_st=working ∧ s=N ∧ s=r)
∨ (r_st=working ∧ s_st=success) ∨ (s_st=working ∧ c_st=failure ∧ r=s+1)
∨ (r_st=working ∧ c_st=failure ∧ r=s+1) ∨ (s_st≠working  ∧ r_st≠working)
```

```
event rcv_file_success extends rcv_file_success
then
  @act5_1 pkt_rcv_t:= clk
end

anticipated event rcv_current_pkt extends
      rcv_current_pkt
then
  @act5_1 pkt_rcv_t:= clk
end
```

```
event tick refines tick
where
  @grd3_1 s_st=working ∧ r=s ⇒clk+1−pkt_tc
      ≤  c_ddl
  @grd3_2 s_st=working ∧ r=s+1 ⇒clk+1−
      pkt_rcv_t≤  c_ddl
  @grd3_3 r_st=working ∧ r=s ⇒clk+1−pkt_tc
      ≤  c_ddl
  @grd3_4 r_st=working ∧ r=s+1 ⇒clk+1−
      pkt_rcv_t≤  pkt_abt_duration
then
  @act1_1 clk:= clk+1
end
```

FIGURE 5.6: Machine $M3$ that Extends $M2$ with Timing Properties Specified In **TR-3** and **TR-4**

5.2.5 Fourth Refinement

Timing is important to synchronize the status of transmitter and receiver in BRP. In $M4$, we want to guarantee that the receiver will abort based on the timing information on its end. In this refinement, we first refine the *anticipated* event *snd_rty* to *convergent* event by introducing variable *rty_cnt* to denote a retry counter. The transmitter will decide to abort if *rty_cnt* reaches the constant *retry_num*. As shown in Figure 5.7, we use $retry\_num - rty\_cnt$ as the variant. The convergent event *snd_rty* always decreases the variant. We also use invariant @*dlf_m4* to verify that the refined machine is not deadlocked by the strengthened guards of *snd_retry*.

```
invariants
@rty_cnt_type rty_cnt∈ ℕ ∧ rty_cnt≤  retry_num
@dlf_m4 (s_st=success ∧ s=0) ∨ (s_st=working ∧ s+1≤ N ∧ r=s+1 ∧ c_st=working)
  ∨ (r_st=working ∧c_st=working ∧ s_st=working ∧ r<N ∧ r=s)
  ∨ (s_st=working ∧ r_st=working ∧ c_st=working)
  ∨ (s_st=working ∧ r_st=working ∧ c_st=failure ∧ rty_cnt<retry_num)
  ∨ (s_st=working ∧ s=N ∧ s=r) ∨ (r_st=working ∧ s_st=success)
  ∨ (s_st=working ∧ c_st=failure ∧ r=s+1 ∧ rty_cnt=retry_num)
  ∨ (r_st=working ∧ c_st=failure ∧ r=s+1 ∧ rty_cnt=retry_num)
  ∨ (s_st≠working  ∧ r_st≠working)
variant
  retry_num−rty_cnt
```

```
convergent event snd_retry extends snd_retry
where
  @grd4_1 rty_cnt<retry_num
then
  @act4_1 rty_cnt:= rty_cnt+1
end

event snd_pkt extends snd_pkt
then
  @act4_1 rty_cnt:= 0
end
```

```
event rcv_abt extends rcv_abt
where
  @grd4_1 rty_cnt=retry_num
end

event snd_abt extends snd_abt
where
  @grd4_1 rty_cnt=retry_num
end
```

FIGURE 5.7: Machine $M4$ that Refines the *anticipated* Event *snd_rty* to *convergent* Event

5.2.6 Fifth Refinement

In $M5$, we refine timing property (5.5b) to timing property (5.6a) and timing property (5.5c) to timing properties (5.6b) and (5.6c). It takes the receiver at least $2 * c\_dly$ time units but at most *retry_ddl* time units before knowing that the transmitter has not sent the new packet. After the resend attempts, the receiver guarantees that the transmitter has aborted. Then it aborts after $2 * c\_dly * (retry\_num + 1)$ time units and within $retry\_ddl * (retry\_num + 1)$ time units. Also, the transmitter might need to resend the packet several times before it receives the acknowledgment from the receiver. In the model shown in Figure 5.8, we replace @$grd3\_2$ of *tick* event in $M3$ with @$grd5\_1$ of *tick* event in $M5$ with the assumption shown in Equation 5.7. @$inv5\_2$ and @$inv5\_3$ are used to present timing properties 5.6. @$inv5\_1$ serves as the auxiliary invariant for

@*inv*5_2. Invariant @*dlf_m*5 is constructed to show that $M5$ is not deadlocked because of the strengthened guard of delay time constraints.

$$Resp(rcv\_current\_pkt, snd\_pkt, 0, retry\_ddl * (rty\_cnt + 1)) \tag{5.6a}$$

$$Resp(rcv\_current\_pkt, snd\_rty, 2 * c\_dly * (rty\_cnt + 1), retry\_ddl * (rty\_cnt + 1)) \tag{5.6b}$$

$$Resp(rcv\_current\_pkt, rcv\_abt, 2 * c\_dly * (retry\_num + 1), retry\_ddl * (retry\_num + 1)) \tag{5.6c}$$

$$\begin{cases} c\_ddl \geq (retry\_num + 1) * rty\_ddl \\ rty\_ddl > 2 * c\_dly \\ pkt\_abt\_duration \geq (retry\_num + 1) * rty\_ddl \end{cases} \tag{5.7}$$

```
invariants
@inv5_1 r_st=working ∧ r=s+1 ⇒clk−pkt_rcv_t≤ rty_ddl ∗ (rty_cnt+1)
@inv5_2 pkt_rcv_t≤ snd_rty_t ∧ r=s+1 ⇒ snd_rty_t− pkt_rcv_t≤ rty_ddl ∗ (rty_cnt+1)
@inv5_3 pkt_rcv_t≤ abt_t ⇒ abt_t− pkt_rcv_t≤ rty_ddl ∗ (retry_num+1)
@dlf_m5 (s_st=success ∧ s=0)∨ (s_st=working ∧ s+1≤ N ∧ r=s+1 ∧ c_st=working)
∨ (r_st=working ∧c_st=working ∧ s_st=working ∧ r<N ∧ r=s)
∨ (s_st=working ∧ r_st=working ∧ c_st=working ∧ clk≥ pkt_rcv_t+ c_dly ∧ snd_rty_t>pkt_rcv_t⇒clk
      ≥ snd_rty_t +2∗c_dly)
∨ (s_st=working ∧ r_st=working ∧ c_st=failure ∧ rty_cnt<retry_num ∧ clk≥ pkt_rcv_t+ c_dly ∧
      snd_rty_t>pkt_rcv_t⇒clk≥ snd_rty_t +2∗c_dly)
∨ (s_st=working ∧ s=N ∧ s=r)∨ (r_st=working ∧ s_st=success)
∨ (s_st=working ∧ c_st=failure ∧ r=s+1 ∧ rty_cnt=retry_num)
∨ (r_st=working ∧ c_st=failure ∧ r=s+1 ∧ rty_cnt=retry_num)
∨ ((s_st=working ∧ r=s ⇒clk+1−pkt_tc≤ c_ddl)∧(s_st=working
∧ r=s+1 ⇒clk+1−pkt_rcv_t≤ rty_ddl ∗ (rty_cnt+1))∧(r_st=working ∧ r=s ⇒clk+1−pkt_tc≤ c_ddl)
∧(r_st=working ∧ r=s+1 ⇒clk+1−pkt_rcv_t≤ rty_ddl ∗ (rty_cnt+1)))
```

```
convergent event snd_rty extends snd_rty
where
  @grd6_1 clk≥ pkt_rcv_t+ c_dly
  @grd6_2 snd_rty_t>pkt_rcv_t⇒clk≥ snd_rty_t
      +2∗c_dly
then
  @act6_1 snd_rty_t:= clk
end
```

```
event tick refines tick
where
  @grd5_1 s_st=working ∧ r=s ⇒clk+1−pkt_tc
      ≤ c_ddl
  @grd5_2 s_st=working ∧ r=s+1 ⇒clk+1−
      pkt_rcv_t≤ rty_ddl ∗ (rty_cnt+1)
  @grd5_3 r_st=working ∧ r=s ⇒clk+1−pkt_tc
      ≤ c_ddl
  @grd5_4 r_st=working ∧ r=s+1 ⇒clk+1−
      pkt_rcv_t≤ rty_ddl ∗ (rty_cnt+1)
then
  @act1_1 clk:= clk+1
end
```

FIGURE 5.8: Machine $M5$ that Extends $M4$ with Timing Properties Specified In **TR-5** and **TR-6**

Table 5.1: Proof Statistics of BRP Case Study

Machine	Generated PO	Automatically Proved	Automatically Proved %
m0	2	2	100
m1	39	39	100
m2	29	29	100
m3	53	53	100
m4	15	12	80
m5	33	28	84.8
m6	13	10	76.9

5.2.7 Sixth Refinement

In the last refinement shown in Figure 5.9, we refine the *anticipated* event *rcv_current_pkt* to *convergent* event with the variant $N-r$. The convergent event *rcv_current_pkt* always decreases the variant. Invariant @*dlf_m6* is used to verify the refined machine is not deadlocked.

```
invariants
@dlf_m6  (s_st=success ∧ s=0)∨ (s_st=working ∧ s+1≤ N ∧ r=s+1 ∧ c_st=working)
∨ (r_st=working ∧c_st=working ∧ s_st=working ∧ r<N ∧ r=s)
∨ (s_st=working ∧ r_st=working ∧ c_st=working ∧ clk≥  pkt_rcv_t+ c_dly ∧ snd_rty_t>pkt_rcv_t⇒clk
      ≥  snd_rty_t +2∗c_dly)
∨ (s_st=working ∧ r_st=working ∧ c_st=failure ∧ rty_cnt<retry_num ∧ clk≥  pkt_rcv_t+ c_dly ∧
      snd_rty_t>pkt_rcv_t⇒clk≥  snd_rty_t +2∗c_dly)
∨ (s_st=working ∧ s=N ∧ s=r)∨ (r_st=working ∧ s_st=success)
∨ (s_st=working ∧ c_st=failure ∧ r=s+1 ∧ rty_cnt=retry_num)
∨ (r_st=working ∧ c_st=failure ∧ r=s+1 ∧ rty_cnt=retry_num)
∨ ((s_st=working ∧ r=s ⇒clk+1−pkt_tc≤  c_ddl)∧(s_st=working
∧ r=s+1 ⇒clk+1−pkt_rcv_t≤  rty_ddl ∗ (rty_cnt+1))∧(r_st=working ∧ r=s ⇒clk+1−pkt_tc≤  c_ddl)
∧(r_st=working ∧ r=s+1 ⇒clk+1−pkt_rcv_t≤  rty_ddl ∗ (rty_cnt+1)))

variant
N−r
```

Figure 5.9: Machine $M6$ that refines *Anticipated* Event *rcv_current_pkt* to *Convergent* Event

5.3 Evaluation and Conclusion

Table 5.1 shows the proof statistics of the BRP model. All the proof obligations related to the timing properties could be discharged automatically. However, the proof obligations related to the relative deadlock freeness cannot be proved automatically because the invariant is too complicated to be proved by the default prover. We proved the invariant by cases manually and verified the model.

Our model of the BRP case study is based on Sarshogh's message passing case study (Sarshogh, 2013) and Sulskus's parallel message-passing case study (Sulskus, 2017). Sarshogh's approach investigated the practicality of their proposed trigger-response pattern and refinement patterns to model real-time properties. Sulskus' approach transformed the single-channel message passing system to a parallel message-passing system with their proposed time interval pattern. Sulskus's time interval approach produced a finite-state space model for model checking purposes. Our approach modeled the real-time system with the unbounded *clock* variable and infinite behavioral traces. Neither Sarshogh nor Sulskus' approach takes the divergence of introduced new events in the refinement step into account. Our approach used the two-step refinement strategy to avoid the divergence of intermediate events and infeasible responses in real-time systems. We encoded the conditions such as conditional convergence and deadlock freeness into Event-B models syntactically. Under weak fairness assumptions and deadlock freeness, we could guarantee the trigger-response properties of the system.

Chapter 6

Bridge the Gap Between Semantics and Formal Models

Event-B is a modeling language that supports modeling discrete systems but lacks explicit support for expressing and verifying timing and liveness properties (Sulskus et al., 2016). In previous chapters, we mainly focused on exploring additional proof obligations and fairness assumptions to treat the real-time trigger-response properties properly. However, there is still a huge gap between the modeler and the domain expert to specify functional requirements with time constraints. Formal specification patterns can be used to facilitate the specification process with reuseable patterns and a pre-defined modeling strategy. In this chapter, we show how to encode real-time properties in Event-B models syntactically. Based on the formalization, we present four real-time specification patterns, namely time response pattern, abort pattern, intermediate pattern and periodic pattern, to support quantitative reasoning about time. The new patterns not only allow modeling of timing properties between trigger and response events, but also allow modeling of the interrupt and reoccurrence behavior. Templates are provided to show how to apply the patterns syntactically in Event-B models. Also, we provide some patterns to refine the real-time specification patterns.

6.1 Modeling Real-time Specification Patterns with Trigger-Response Properties

In this section, we provide a list of patterns partitioned into four main categories, namely time response patterns, abort patterns, intermediate patterns and periodic patterns. We do not define real-time specification patterns for universality pattern and absence pattern mentioned in Section 2.4.1 as these two patterns are not constrained by timing properties. We present the patterns with Event-B formalization.

6.1.1 Time Response Patterns

The time response patterns are used to specify the time interval between the events that satisfy trigger-response ordering property, which describes the behavior that when the trigger event occurs, the corresponding response event will occur with the specified time interval. In Chapter 3 and Chapter 4, we mainly explore conditions required to model a real-time system from the semantics perspective. For example, In Theorem 3.14, we provide additional conditions with which the Event-B machine satisfies trigger-response properties. In this section, we show how to encode these conditions in Event-B models syntactically in the way that standard Event-B proof obligations are defined. Based on Theorem 3.14, we define the trigger-response ordering property defined in Definition 6.1 with Event-B proof obligations. Condition 1 requires that either trigger event or response event is enabled in the machine, which satisfies Condition 1 in Theorem 3.14. In this setting, we assume that there are no intermediate events H between T and R, thus Condition 2 requires that the trigger event enables the response event and disables itself, which satisfy Condition 2 in Theorem 3.14. Condition 3 requires that the enabledness of response events are preserved by each $e \in E \setminus (T \cup R)$, which satisfies Condition 4 in Theorem 3.14.

Definition 6.1 (Trigger-Response Ordering Property)**.** Given an Event-B machine M with events E and invariants $I(v)$, a trigger-response pair has the form $TR(T, R)$ where $T \subseteq E$ are trigger events, $R \subseteq E$ are response events, and $T \cap R = \varnothing$. M is defined to satisfy the trigger-response ordering property $TR(T, R)$ when:

1. $G_R(v) \Rightarrow \neg G_T(v)$
2. $t \in T \wedge I(v) \wedge G_t(v) \wedge S_t(v, v') \Rightarrow \neg G_t(v') \wedge G_R(v')$
3. $e \in E \setminus (T \cup R) \wedge I(v) \wedge G_e(v) \wedge S_e(v, v') \wedge G_R(v) \Rightarrow G_R(v')$

Figure 6.1 depicts the syntactic formalization we use that extends the trigger-response pair $TR(T, R)$ to the timing property $Resp(T, R, w, d)$ with the time response pattern, where $TR(T, R)$ satisfies the trigger-response ordering property defined in Definition 6.1, and w and d define the delay and deadline time interval between trigger and response events. We asserted that the behaviors of a machine with $Resp(T, R, w, d)$ should satisfy two properties: 1) the number of *Tick* events between trigger events T and response events R is bounded by the delay time w and deadline time d, and 2) the response event eventually occurs after the trigger event, and the trigger event does not recur within the trigger-response pair to avoid the recurring delay of response events. In the formalization, a new variable *clk* is used to present the global clock, and the event *Tick* is used to proceed the clock. The timestamp variable $\tau_T \in 0..clk$ and $\tau_R \in 0..clk$ are set by the before-after predicate in the trigger and response events, respectively. Additional constraints relating to τ_T and τ_R are imposed on R and *Tick* event to model the time interval $[w, d]$ between trigger and response events. Guard $G_R(v) \Rightarrow clk + 1 - \tau_T \leq d$ of the *Tick* event constrains the global clock not to tick when the response event is about to miss its deadline. Invariant @*rsp*_2 and @*rsp*_3 show that the difference between timestamps of trigger events and response events are bounded by the time interval $[w, d]$, and Invariant @*rsp*_1 serves as an auxiliary invariant to prove Invariant @*rsp*_2. A parameterized version of proof that proves that Invariant @*rsp*_3 is preserved by all the events of the machine with timing encoded is provided in Theorem 8.4 in Chapter 8.

In Theorem 3.14, weak fairness assumption on response events is needed to guarantee the feasible occurrence of response events. We also assume that the *Tick* event is weakly fair so that when it is continuously enabled, it is assumed to fire infinitely often. However, Event-B is not concerned with fairness specifications. In the syntactic formalization shown in Figure 6.1, we assume that event R and event *Tick* are weakly fair.

```
invariants
@rsp_1 $G_R(v) \Rightarrow clk - \tau_T \leq d$
@rsp_2 $\tau_T < \tau_R \Rightarrow \tau_R - \tau_T \geq w$
@rsp_3 $\tau_T < \tau_R \Rightarrow \tau_R - \tau_T \leq d$

event T
where
  $G_T(v)$
then
  $v := S_T(v, v')$
  $\tau_T := clk$
end
```

```
event R
where
  $G_R(v)$
  @grd1 $clk \geq \tau_T + w$
then
  $v := S_R(v, v')$
  $\tau_R := clk$
end

event Tick
where
  @grd1 $G_R(v) \Rightarrow clk + 1 - \tau_T \leq d$
then
  $clk := clk + 1$
end
```

FIGURE 6.1: Syntactic Formalization of Timing Property $Resp(T, R, w, d)$ with Time Response Pattern

6.1.2 Abort Patterns

The abort patterns are used to describe the time interval between the trigger-response pair, which could be interrupted by the interrupt event INT. This pattern asserts that when the trigger event occurs, the corresponding response event will occur within the specified time interval unless some interrupting event occurs in between. We define the trigger-interrupt-response property as Definition 6.2. The conditions are the same as Definition 6.1 except Condition 2 and Condition 4. Condition 2 requires that the trigger event enables not only the response events but also the interrupt events. The interrupt cannot occur before the trigger event. Condition 4 requires that the interrupt event enables some predicate $Q(v')$. We assume that the system is weakly fair with respect to the response events so that the trigger event is followed eventually by a response event. Figure 6.2 presents the formalization that extends (T, R, INT) to the timing property

$Abt(T, R, INT, w, d)$ with the abort pattern. In the formalization, the predicate $\neg Q(v')$ is added on the guard of response event, and *Tick* event. The occurrence of the interrupt event will remove the time constraints imposed on response and *Tick* event. Invariant @*abt*_2 and @*abt*_3 show that the difference between timestamps of trigger events and response events are bounded by the time interval $[w, d]$ provided the interrupt did not occur in between. Invariant @*abt*_1 serves as an auxiliary invariant to prove Invariant @*abt*_3.

Definition 6.2 (Trigger-Interrupt-Response Property)**.** Given an Event-B machine M with events E and invariants $I(v)$, a trigger-interrupt-response pair has the form (T, R, INT) where $T \subseteq E$ are trigger events, $R \subseteq E$ are response events, INT are interrupt events. M is defined to satisfy the trigger-interrupt-response property when:

1. $G_R(v) \Rightarrow \neg G_T(v)$
2. $t \in T \wedge I(v) \wedge G_t(v) \wedge S_t(v, v') \Rightarrow \neg G_t(v') \wedge G_{INT}(v') \wedge G_R(v')$
3. $e \in E \setminus (T \cup R) \wedge I(v) \wedge G_e(v) \wedge S_e(v, v') \wedge G_R(v) \Rightarrow G_R(v')$
4. $i \in INT \wedge I(v) \wedge G_i(v) \wedge S_i(v, v') \Rightarrow Q(v')$

```
invariants
@abt_1 $G_R(v) \wedge \neg Q(v) \Rightarrow clk - \tau_T \leq d$
@abt_2 $G_R(v) \wedge \neg Q(v) \wedge \tau_T < \tau_R \Rightarrow \tau_R - \tau_T \geq w$
@abt_3 $G_R(v) \wedge \neg Q(v) \wedge \tau_T < \tau_R \Rightarrow \tau_R - \tau_T \leq d$
```

```
event T
where
    $G_T(v)$
then
    $v := S_T(v, v')$
    $\tau_T := clk$
end

event INT
where
  $G_{INT}(v)$
then
  $v := S_{INT}(v, v')$
end
```

```
event R
where
    $G_R(v) \wedge \neg Q(v) \Rightarrow clk \geq \tau_T + w$
then
    $v := S_R(v, v')$
    $\tau_R := clk$
end

event Tick
where
  $G_R(v) \wedge \neg Q(v) \Rightarrow clk + 1 - \tau_T \leq d$
then
  $clk := clk + 1$
end
```

FIGURE 6.2: Syntactic Formalization of Timing Property $Abt(T, R, INT, w, d)$ with Abort Pattern

6.1.3 Intermediate Patterns

The intermediate patterns are used to describe the time interval between the cases that the trigger events do not lead to response events directly. This pattern asserts that the occurrence of trigger events enables the intermediate events H, which could occur sufficiently often before the response event occurs. Provided that the intermediate events are convergent, the repeated occurrence of intermediate events eventually enables the response events. We define the trigger-intermediate-response property as Definition 6.3. In the definition, we use $G_H(v)$ to denote the disjunction of all $G_i(v)$ with $i \in H$. Condition 2 requires the occurrence of trigger events that enables one of the intermediate events or response events. Condition 3 requires that all the other events preserve the enabledness of intermediate events or response events. We assume that the system is weakly fair with respect to intermediate events and response events. Condition 4 guarantees that once a response event is enabled, it cannot be disabled by any event other than a response event, which avoids the scenario that intermediate events and response events are enabled alternatively but never get executed under weak fairness assumption. Also, Condition 5 requires that the intermediate events should converge towards a response event being enabled. Weak fairness assumptions on the intermediate events guarantee that the intermediate events get executed sufficiently often to lead to the response event being enabled. Then weak fairness assumptions on the response events guarantee that an enabled response event eventually gets executed.

Definition 6.3 (Trigger-Intermediate-Response Property)**.** Given an Event-B machine M with events E and invariants $I(v)$, a trigger-intermediate-response pair has the form (T, H, R) where $T \subseteq E$ are trigger events, $R \subseteq E$ are response events, H are intermediate events. Given each $t \in T$ and $r \in R$, M is defined to satisfy the trigger-intermediate-response property when:

1. $G_R(v) \Rightarrow \neg G_T(v)$;
2. $I(v) \wedge G_t(v) \wedge S_t(v, v') \Rightarrow \neg G_t(v') \wedge (G_H(v') \vee G_R(v'))$;
3. $e \in E \setminus (T \cup R) \wedge I(v) \wedge G_e(v) \wedge S_e(v, v') \wedge G_H(v) \Rightarrow (G_H(v') \vee G_R(v'))$;
4. $e \in E \setminus (T \cup R) \wedge I(v) \wedge G_e(v) \wedge S_e(v, v') \wedge G_R(v) \Rightarrow G_R(v')$;
5. The intermediate events H are conditional convergent under the condition that response events R are disabled;

Figure 6.3 shows the formalization that extends the trigger-intermediate-response property to the timing property $Interm(T, H, R, w, d)$ with the intermediate pattern. Based on Theorem 3.14, the intermediate events H are conditional convergent to response events R. Thus two versions of the intermediate events are included in the machine to encode *conditional convergence* syntactically in Event-B models. As shown in Figure 6.3, the H_cov event is convergent when the response events are disabled. Event H, which has the guard requires response event is enabled, is not marked as convergent. A new variable inc, increased by the intermediate event H, is added to the machine. The variant is decreased by the intermediate events H. With the VAR and NAT proof obligations, we can prove that intermediate events are convergent towards disabled response events. In the formalization, we assume that the machine is weakly fair with respect to H, R and $Tick$ events.

```
**constants** max
**variables** inc
**invariants**
@int_1 $G_R(v) \Rightarrow clk - \tau_T \leq d$
@int_2 $G_R(v) \wedge \tau_T < \tau_R \Rightarrow \tau_R - \tau_T \geq w$
@int_3 $G_R(v) \wedge \tau_T < \tau_R \Rightarrow \tau_R - \tau_T \leq d$
**variant** $max - inc$
```

```
convergent event H_cov
where
$G_H(v)$
$\neg G_R(v)$
then
$v := S_H(v, v')$
$inc := inc + 1$
end

event H
where
$G_H(v)$
$G_R(v)$
then
$v := S_H(v, v')$
end
```

```
event T
where
  $G_T(v)$
then
  $v := S_T(v, v')$
  $\tau_T := clk$
end

event R
where
  $G_R(v) \Rightarrow clk \geq \tau_T + w$
then
  $v := S_R(v, v')$
  $\tau_R := clk$
end

event Tick
where
$G_R(v) \Rightarrow clk + 1 - \tau_T \leq d$
then
$clk := clk + 1$
end
```

FIGURE 6.3: Syntactic Formalization of Timing Property $Interm(T, H, R, w, d)$ with Intermediate Pattern

6.1.4 Periodic Patterns

The periodic patterns are used to present the time interval in which the event should occur at least once. We define the repeatable event property in Definition 6.4. Condition 1 requires that events T are initially enabled. Condition 2 requires that the enabledness of T is preserved by all the events in the model. We call T as repeatable events provided the two conditions hold. Figure 6.4 shows the formalization that extends the repeatable event T to timing property $Prd(T, w, d)$ with periodic pattern. In the formalization, the timestamp variable τ_{T}_c denotes the current timestamp that T occurs. Additional constraints are imposed on T and *Tick* event to model the time interval $[w, d]$ between two T events. To describe the time interval with invariants, we add another timestamp variable τ_{T}_p to denote the timestamp of the previous T event. τ_{T}_p is set to τ_{T}_c by the before-after predicate in T. Invariant *prd*_2 and *prd*_3 show that the time difference between two adjacent T events are bounded by w and d. Invariant @*prd*_1 serves as an auxiliary invariant to prove Invariant @*prd*_3.

Definition 6.4 (Repeatable Property)**.** Given an Event-B machine M with events E and invariants $I(v)$, the event $T \subseteq E$ are repeatable events. M satisfies repeatable property when:

1. $I(v) \wedge S_{init}(v, v') \Rightarrow G_T(v')$
2. $e \in E \wedge I(v) \wedge G_e(v) \wedge S_e(v, v') \wedge G_T(v') \Rightarrow G_T(v')$

```
invariants
@prd_1 $G_T(v) \Rightarrow clk - \tau_T \leq d$
@prd_2 $\tau_T\_p < \tau_T\_c \Rightarrow \tau_T\_c - \tau_T\_p \geq w$
@prd_3 $\tau_T\_p < \tau_T\_c \Rightarrow \tau_T\_c - \tau_T\_p \leq d$
```

```
event T
where
    $G_T(v)$
    $clk \geq \tau_T\_c + w$
then
    v:= $S_T(v, v')$
    $\tau_T\_c := clk$
    $\tau_T\_p := \tau_T\_c$
end
```

```
event Tick
where
    $G_T(v) \Rightarrow clk + 1 - \tau_T\_c \leq d$
then
    $clk := clk + 1$
end
```

FIGURE 6.4: Syntactic Formalization of Timing Property $Prd(T, w, d)$ with Periodic Pattern

6.2 Patterns to Refine Real-time Properties

Lemma 4.5 and Lemma 4.6 provide additional conditions to refine abstract timing properties into sequential or alternative sub-timing properties from semantics perspective. In this section, we present the patterns to refine abstract timing properties into concrete timing properties with the syntactic formalization in Event-B. Each pattern is explained with the cases of refining four real-time properties mentioned in Section 6.1.

6.2.1 Sequential Refinement Pattern

The sequential refinement pattern refines abstract timing properties into a sequence of sub-timing properties. Based on Lemma 4.5, the sum of the delay and deadline duration of sub-timing properties should be consistent with the abstract delay and deadline duration.

Take the timing property (6.1) as an example, we introduce intermediate event H between the trigger-response pair $TR(T, R)$ such that (T, H) and (H, R) satisfy the trigger-response ordering property defined in Definition 6.1. We use timing properties (6.2) to show the delay and deadline time constraints of two trigger response pairs. Figure 6.5 shows the time diagram that refines abstract timing property (6.1) with concrete timing properties (6.2).

$$Resp(T, R, w, d) \tag{6.1}$$

$$\begin{cases} Resp(T, H, w1, d1) \\ Resp(H, R, w2, d2) \end{cases} \tag{6.2}$$

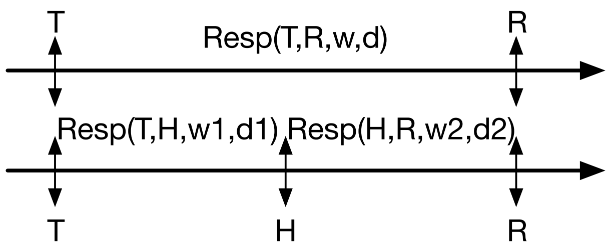

FIGURE 6.5: Time Diagram of Refining Abstract Timing Property (6.1) with Concrete Timing Properties (6.2)

Figure 6.6 depicts the syntactic formalization of the sequential refinement pattern applied to refine the machine shown in Figure 6.1. Invariants @$rspr\_1$ and @$rspr\_2$, as well as events T, R, H and $Tick$ follows the time response pattern shown in Figure 6.1. The additional axiom @rsp_axm is added based on Lemma 4.5 to guarantee the consistency between the timing properties. In the refinement, we replace @$grd1$ of $Tick$ event with @$grd1$ and @$grd2$ in the refinement. The GRD proof obligation of $Tick$ event could be discharged with the following proof:

```
axioms
@rsp_axm $w \leq w1 + w2 \wedge d1 + d2 \leq d$
invariants
@rspr_1 $\tau_T < \tau_H \Rightarrow \tau_H - \tau_T \geq w1 \wedge \tau_H - \tau_T \leq d1$
@rspr_2 $\tau_H < \tau_R \Rightarrow \tau_R - \tau_H \geq w2 \wedge \tau_R - \tau_H \leq d2$
@dlf $G_T(v) \vee (G_R(v) \wedge clk \geq \tau_T + w) \vee ((G_R(v) \Rightarrow clk + 1 \leq \tau_T + d)) \Rightarrow$
$G_T(v) \vee (G_H(v) \wedge clk \geq \tau_T + w1) \vee (G_R(v) \wedge clk \geq \tau_H + w2) \vee$
$((G_R(v) \Rightarrow clk + 1 \leq \tau_H + d2) \wedge (G_H(v) \Rightarrow clk + 1 \leq \tau_T + d1))$
```

```
event T
where
  $G_T(v)$
then
  $v := S_T(v, v')$
  $\tau_T := clk$
end

event H
where
  $G_H(v)$
  $clk \geq \tau_T + w1$
then
  $v := S_H(v, v')$
  $\tau_H := clk$
end
```

```
event R
where
  $G_R(v)$
  @grd1 $clk \geq \tau_H + w2$
then
  $v := S_R(v, v')$
  $\tau_R := clk$
end

event Tick
where
  @grd1 $G_H(v) \Rightarrow clk + 1 - \tau_T \leq d1$
  @grd2 $G_R(v) \Rightarrow clk + 1 - \tau_H \leq d2$
then
$clk := clk + 1$
end
```

FIGURE 6.6: Refining Time Response Patterns with Sequential Sub-Timing Properties

Proof. We want to prove that $I(v) \wedge J(v, w) \wedge G_{R'}(w) \Rightarrow G_R(v)$. Assume $G_R(v)$, we have to show $clk + 1 \leq \tau_T + d$:

$$\begin{aligned}
& clk + 1 \\
\leq & \{@grd2 : clk + 1 - \tau_H \leq d2\} \\
& \tau_H + d2 \\
\leq & \begin{cases} \tau_T + d2 \ \{\tau_H \leq \tau_T\} \\ \tau_T + d1 + d2 \ \{\tau_T \leq \tau_H \wedge @rspr_1 : \tau_H - \tau_T \leq d1\} \end{cases} \\
\leq & \begin{cases} \tau_T + d \ \{@rsp\_axm : d1 + d2 \leq d\} \\ \tau_T + d \ \{@rsp\_axm : d1 + d2 \leq d\} \end{cases}
\end{aligned}$$

□

Based on Lemma 4.2, we require that the refined machine is deadlock-free relative to the abstract machine. So in the formalization, we construct Invariant *dlf* to verify the deadlock freeness of the refined machine. Also, we assume that the refined machine is weakly fair to H, R and $Tick$ event. If multiple intermediate events exist between the trigger and response events, we require them to be *conditional convergent* so that the response events are feasible. The formalization of conditional convergent events is shown in Figure 6.3.

This sequential refinement pattern could also be applied to refine the timing properties modeled with abort, intermediate and periodic patterns.

Given the timing property (6.3) modeled with the abort pattern, the interrupt events INT could occur between T and R nondeterministically. As shown in Figure 6.7, when the abstract trigger-response pair $TR(T, R)$ is refined into two sequential trigger-response pairs, namely (T, H) and (H, R), the INT event not only could occur between T and H nondeterministically, but also could occur between H and R nondeterministically. Thus the timing property (6.3) could be refined to sub-timing properties (6.4) provided the sum of the delay and deadline duration of sub-timing properties is consistent with the abstract delay and deadline duration.

$$Abt(T, R, INT, w, d) \tag{6.3}$$

$$\begin{cases} Abt(T, H, INT, w1, d1) \\ Abt(H, R, INT, w2, d2) \end{cases} \tag{6.4}$$

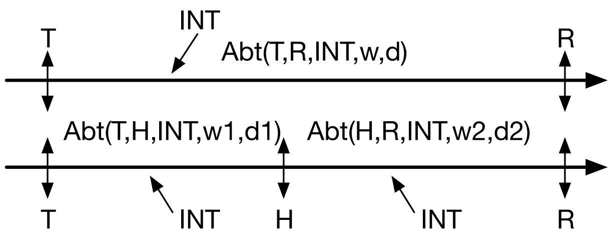

FIGURE 6.7: Time Diagram of Refining Timing Properties (6.3) with Sequential Sub-Timing Properties (6.4)

Figure 6.8 depicts the time diagram of refining timing properties (6.5) with sequential sub-timing properties (6.6). Based on the refinement pattern, relations between abstract and concrete timing properties specified in Equation 6.7 need to specified to keep the consistency.

$$Interm(T, H, R, w, d) \tag{6.5}$$

$$\begin{cases} Resp(T, H, w1, d1) \\ Prd(H, w3, d3) \\ Resp(H, R, w2, d2) \end{cases} \tag{6.6}$$

$$\begin{cases} w \leq w1 + w2 + w3 * (n-1) \\ d1 + d2 + d3 * (n-1) \leq d \end{cases} \tag{6.7}$$

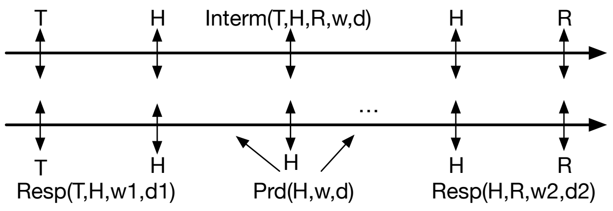

FIGURE 6.8: Time Diagram of Refining Timing Properties (6.5) with Sequential Sub-Timing Properties (6.6)

Figure 6.9 shows the time diagram that refines timing property (6.8) with sequential sub-timing properties (6.9). Given that $w \leq w1 + w2$ and $d1 + d2 \leq d$, the consistency of timing properties in different refinement levels is preserved.

$$Prd(T, w, d) \tag{6.8}$$

$$\begin{cases} Resp(T, R, w1, d1) \\ Resp(R, T, w2, d2) \end{cases} \tag{6.9}$$

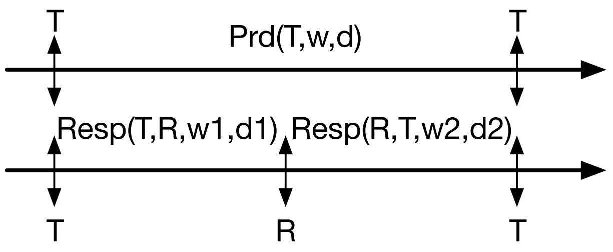

FIGURE 6.9: Time Diagram of Refining Timing Properties (6.8) with Sequential Sub-Timing Properties (6.9)

6.2.2 Alternative Refinement Pattern

The alternative refinement pattern refines abstract timing properties into alternative sub-timing properties. Together with the sequential refinement pattern, the alternative refinement pattern could be used to resolve different levels of nondeterminism of intermediate events. For example, Figure 6.10 shows the time diagram that refines timing property (6.1) to alternative timing properties (6.10). In the refined machine, either $H1$ event or $H2$ event occurs following the trigger event T. Then $H1$ and $H2$ would lead to different response events $R1$ and $R2$ respectively. Based on Lemma 4.6, equation 6.11 is required to preserve the consistency of timing properties.

$$\begin{cases} Resp(T, H1, w1, d1) \\ Resp(H1, R1, w2, d2) \\ Resp(T, H2, w3, d3) \\ Resp(H2, R2, w4, d4) \end{cases} \tag{6.10}$$

$$\begin{cases} w \leq (w_1 + w_2) \wedge w \leq (w_3 + w_4) \\ (d_1 + d_2) \leq d \wedge (d_3 + d_4) \leq d \end{cases} \tag{6.11}$$

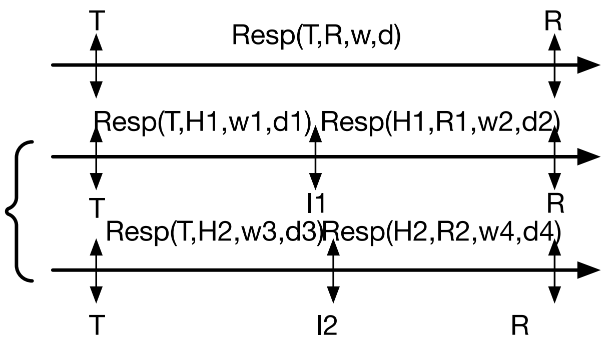

FIGURE 6.10: Time Diagram of Refining Timing Properties (6.1) with Alternative Sub-Timing Properties (6.10)

Figure 6.11 depicts the syntactic formalization of the alternative refinement pattern used to refine the machine shown in Figure 6.1. Additional axioms @alt_axm1 and @alt_axm2 is used to keep the consistency of timing property based on Lemma 4.6. In the refinement, we replace @$grd1$ of *Tick* event with @$grd1$, @$grd2$, @$grd3$ and @$grd4$ in the refinement. The GRD proof obligation of *Tick* event could be discharged with the following proof:

Proof. We want to prove that $I(v) \wedge J(v, w) \wedge G_{R'}(w) \Rightarrow G_R(v)$. Assume $G_R(v)$, we have to show $clk + 1 \leq \tau_T + d$:

$$
\begin{aligned}
& clk + 1 \\
\leq & \begin{cases} \tau_{H1} + d2 \ \{G_{H1}(v) \text{ and } @grd3 : G_R(v) \Rightarrow clk + 1 - \tau_{H1} \leq d2\} \\ \tau_{H2} + d4 \ \{G_{H2}(v) \text{ and } @grd4 : G_R(v) \Rightarrow clk + 1 - \tau_{H2} \leq d4\} \end{cases} \\
\leq & \begin{cases} \begin{cases} \tau_T + d2 \ \{\tau_{H1} \leq \tau_T\} \\ \tau_T + d1 + d2 \ \{\tau_T \leq \tau_{H1}, @alt\_1 : \tau_H - \tau_T \leq d1\} \end{cases} \\ \begin{cases} \tau_T + d4 \ \{\tau_{H2} \leq \tau_T\} \\ \tau_T + d3 + d4 \ \{\tau_T \leq \tau_{H2}, @alt\_3 : \tau_{H2} - \tau_T \leq d3\} \end{cases} \end{cases} \\
\leq & \begin{cases} \begin{cases} \tau_T + d \ \{@alt\_axm1 : d1 + d2 \leq d\} \\ \tau_T + d \ \{@alt\_axm1 : d1 + d2 \leq d\} \end{cases} \\ \begin{cases} \tau_T + d \ \{@alt\_axm2 : d3 + d4 \leq d\} \\ \tau_T + d \ \{@alt\_axm2 : d3 + d4 \leq d\} \end{cases} \end{cases}
\end{aligned}
$$

□

In the formalization, we assume that the refined machine is weakly fair with respect to $H1$, $H2$, $R1$, $R2$ and *Tick* event. Also, we need to check that the refined machine is relative deadlock-free to the abstract machine, which could be encoded as invariants in a machine.

6.3 Applying Real-time Specification Patterns and Refinement Patterns

In this section, we outline how the patterns could be used to develop real-time systems. Suppose the system is extended with the *clk* variable and *Tick* event. Firstly, events identified as trigger events, response events, intermediate events, abort events, and periodic events should be presented with the guards satisfying the corresponding conditions in the patterns. Next, the pattern can be adapted by initializing the real-time properties with the pattern template. For example, new variables presenting the timestamps of trigger and response events will be added in the time response pattern. Then invariants and additional guards should be added based on the template to present the delay and deadline between the trigger-response pair. Finally, the refinement patterns mentioned above could be used to refine the timing properties to sub-timing properties. In Chapter 7, we show how to use these patterns to model time constraints in pacemakers.

```
axioms
@alt_axm1 $w \leq w1 + w2 \wedge d1 + d2 \leq d$
@alt_axm2 $w \leq w3 + w4 \wedge d3 + d4 \leq d$
invariants
@alt_1 $\tau_T < \tau_{H1} \Rightarrow \tau_{H1} - \tau_T \geq w1 \wedge \tau_{H1} - \tau_T \leq d1$
@alt_2 $\tau_{H1} < \tau_R \Rightarrow \tau_R - \tau_{H1} \geq w2 \wedge \tau_R - \tau_{H1} \leq d2$
@alt_3 $\tau_T < \tau_{H2} \Rightarrow \tau_{H2} - \tau_T \geq w3 \wedge \tau_{H2} - \tau_T \leq d3$
@alt_4 $\tau_{H2} < \tau_R \Rightarrow \tau_R - \tau_{H2} \geq w4 \wedge \tau_R - \tau_{H2} \leq d4$
```

```
event T
where
  $G_T(v)$
then
  $v := S_T(v, v')$
  $\tau_T := clk$
end

event H1
where
  $G_{H1}(v)$
  $clk \geq \tau_T + w1$
then
  $v := S_{H1}(v, v')$
  $\tau_{H1} := clk$
end

event H2
where
  $G_{H2}(v)$
  $clk \geq \tau_T + w3$
then
  $v := S_{H2}(v, v')$
  $\tau_{H2} := clk$
end
```

```
event R1
where
  @grd1 $clk \geq \tau_{H1} + w2$
then
  $v := S_{R1}(v, v')$
  $\tau_{R1} := clk$
end

event R2
where
  @grd1 $clk \geq \tau_{H2} + w4$
then
  $v := S_{R2}(v, v')$
  $\tau_{R2} := clk$
end

event Tick
where
  @grd1 $G_{H1}(v) \Rightarrow clk + 1 - \tau_T \leq d1$
  @grd2 $G_{H2}(v) \Rightarrow clk + 1 - \tau_T \leq d3$
  @grd3 $G_R(v) \Rightarrow clk + 1 - \tau_{H1} \leq d2$
  @grd4 $G_R(v) \Rightarrow clk + 1 - \tau_{H2} \leq d4$
then
$clk := clk + 1$
end
```

FIGURE 6.11: Refining Time Response Patterns with Alternative Sub-Timing Properties

6.4 Conclusion

Based on the trigger-response properties in Event-B models, we presented four real-time specification patterns with syntax to model real-time properties in real-world cases. Our patterns enable the modelers to express real-time properties that can be verified with the theorem-proving techniques. Also, we provided the refinement patterns to refine abstract timing properties modeled with the real-time specifications to concrete timing properties. Proofs were provided to prove the consistency of timing properties.

We imposed weak fairness assumptions on intermediate events, response events, and the *Tick* event to ensure that they do get executed sufficiently often when enabled. However, our approach does not rule out several rounds of trigger-response events occurring within one clock tick. One solution to rule out these behaviors is to force the lower bound of real-time trigger-response property $w > 0$. In the cases that there is no lower bound of delay time for trigger-response properties, some stronger fairness restrictions such as finitary fairness (Alur and Henzinger, 1998) or bounded fairness (Dershowitz et al., 2003) can be used to guarantee that *Tick* event gets the chance to proceed.

Chapter 7

Pacemaker Case Study

The safety issue of medical devices has received considerable attention in recent years. There is a demand for formal approaches and tools to specify and verify the safety properties in medical devices (Jiang et al., 2012). Pacemakers are electronic devices implanted in the body to regulate the heartbeat, which should deliver therapies according to misbehavior of the heart within fixed time constraints (Barold et al., 2010). Missing the stimulus deadlines could be life-threatening. In this chapter, we use a dual-chamber pacemaker under DDD mode (Scientific, 2007) as a case study to demonstrate the usage of our proposed real-time specification patterns in Chapter 6. Models are proved with the Rodin tool.

7.1 Pacemaker System Overview

Injuries and coronary artery disease might lead to heart arrhythmia that causes the heart to beat too fast, too slow, or irregularly. The cardiac pacemaker is designed to deliver timely electrical stimuli over the leads with electrodes in contact with the heart to regulate the heart beat (Barold et al., 2010). Pacemakers use leads to sense the Atrial Sense (AS) and Ventricular Sense (VS) generated by local heart tissues and deliver Atrial Pacing (AP) and Ventricular Pacing (VP) if irregular heart behavior is detected. Understanding the complexity of timing cycles in pacemakers is essential as missing stimulus deadlines could cause heart failure and even loss of life. In this effort, we apply a stepwise development approach to model the random heart model and pacemaker model with Event-B formalism. The real-time specification patterns are used to model different timing cycles in the pacemakers. Theorem proving is used to verify the timing properties of the proposed system.

The pacemaker function mode is characterized by a three-letter code that describes the chamber paced, chamber sensed, and mode of response, respectively. For example,

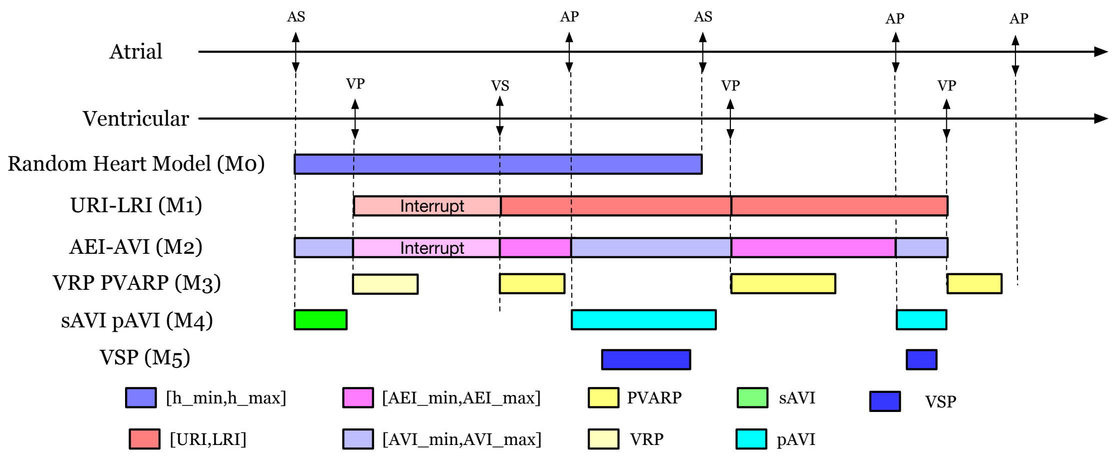

FIGURE 7.1: Timing Cycles of Pacemaker under DDD mode

the code DDD describes that the pacemaker is pacing and sensing in both atrium and ventricle; the pacing is inhibited by the sensed atrial and ventricular event in the atrial channel. However, the pacing is only inhibited by the sensed ventricular event only in the ventricular channel. The pacing is triggered by the sensed atrial activity in the ventricular channel (Barold et al., 2010). We use the pacemaker under DDD mode as a case study to illustrate the usage of real-time specification patterns in Event-B models. There are five primary timing cycles of a DDD pacemaker, namely Lower Rate Interval (LRI), Upper Rate Interval (URI), Ventricular Refractory Period (VRP), Atrioventricular Interval (AVI) and Postventricular Refractory Period (PVARP) (Barold et al., 2010). LRI and URI define the longest and shortest interval between VP or VS and the succeeding VP without intervening VS. VRP defines the interval during which a new LRI cannot be initiated after a ventricular event. AVI defines the interval between AS or AP, and the scheduled delivery of VP. PVARP defines the time interval after the occurrence of a VS or VP, during which the AP cannot initiate a new AVI. These five timing cycles can be used to derive two other timing cycles, namely Atrial Escape Interval (AEI) and Total Atrial Refractory Period (TARP). AEI defines the interval between VS or VP and the succeeding AP providing there is no AS in between. TARP defines the time interval between two AP. In real cases of heart pacing, crosstalk could occur when the pacemaker stimulus in one chamber is sensed in the other chamber. The time interval ventricular safety pacing (VSP) during which the sensed ventricular signal does not inhibit the DDD pacemaker could be introduced in a refinement step to prevent the consequence of crosstalk.

Figure 7.1 shows our refinement strategy of timing cycles of pacemaker between atria and ventricle events. The time intervals in Figure 7.1 could be interrupted. For example, the time interval $[URI, LRI]$ between two adjacent VP events could be interrupted by the nondeterministic VS event, and the time interval between the VS and succeeding VP should be bounded by $[URI, LRI]$. We show our refinement strategy as follows to formalize the timing cycles of a pacemaker under DDD mode.

M_0 specifies the random heart model for atrial channel and ventricular channel.

M_1 introduces URI and LRI to maintain the ventricular rate within a certain threshold.

M_2 formalizes AEI and AVI timing properties by introducing AP into the model.

M_3 formalizes VRP and PVARP by putting delay time constraints on VS, VP and AP that these three events cannot be initiated.

M_4 formalizes the sensed AV interval (sAVI) and paced AV interval (pAVI) based on different types of atrial events that initiate AVI.

M_5 introduces the VSP time interval during which the VP would not be triggered.

7.2 Modeling the Pacemaker

7.2.1 Random Heart Model

In the most abstract level M_0, the Random Heart Model (RHM) is used to describe the nondeterministic heart behaviors. We use XS to present the sensed events in either the atrial or ventricular channel. In the model, the heart generates AS or VS periodically within the $[h\_min, h\_max]$ time interval. The RHM covers both normal and abnormal behaviors by setting the time interval to $[0, \infty]$. In later refinements, the pacing events from pacemakers can be introduced to the model to regulate the heart rate. We begin by modeling the periodic XS events with the periodic pattern $Prd(XS, h\_min, h\_max)$ in our initial model formalized in Figure 7.2. t_xs_p and t_xs_c are used to refer to the time at which previous and current XS event occurs, where $XS = \{AS, VS\}$. The guard $clk \geq t\_xs\_c + h\_min$ guarantees that the next XS event should wait h_min time units before it is enabled. Moreover, $@grd1$ on $tick$ event guarantees that the global clock does not tick when XS is about to miss the deadline. $inv0\_2$ shows the time interval between t_xs_p and t_xs_c is bounded by h_min and h_max. $@inv0\_1$ is the auxiliary invariant to prove $inv0\_2$. In our settings, h_min could be set small enough and h_max could be set large enough to present the irregular behavior of hearts. In the refinement, the pacemaker could sense the irregular behavior and pace the heart to behave regularly.

7.2.2 URI and LRI

The arrhythmias are mainly categorized into two categories, namely tachycardia and bradycardia. Tachycardia describes the abnormally fast heart rate, and bradycardia features a slow heart rate. The pacemaker treats tachycardia and bradycardia by maintaining the heart rate within a time interval, which is not too slow nor too fast. In the first refinement, we first introduce URI and LRI as the time interval.

```
invariants
@inv0_1 clk−t_xs_c≤ h_max
@inv0_2 t_xs_p<t_xs_c ⇒ (t_xs_c−t_xs_p≥ h_min ∧t_xs_c−t_xs_p≤h_max)
```

```
event XS
where
  @grd1 clk≥  t_xs_c+h_min
then
  @act1 t_xs_p:= t_xs_c
  @act2 t_xs_c:= clk
end
```

```
event tick
where
  @grd1 clk+1−t_xs_c≤ h_max
then
  @act1 clk:= clk+1
end
```

FIGURE 7.2: Model Random Heart Model with Periodic Pattern $Prd(XS, h\_min, h\_max)$

Figure 7.3 shows the formalization that model URI and LRI with abort pattern, time response pattern and periodic pattern. The time interval $[URI, LRI]$ between two adjacent VP events could be interrupted by the nondeterministic VS event. The time interval between VS and the succeeding VP should also be bounded by $[URI, LRI]$. The event VS_Trig models the point that VS occurs when no VP has ever occurred. Event VS_Abt models the point that VS occurs between two VP events. We then use VP_Trig and VP_Resp to distinguish the previous and current VP events. $@inv1\_4$ is used to guarantee that a ventricle pace can only occur at least URI and at most LRI after a ventricle event. $@inv1\_2$, $@inv1\_2$ and $@inv1\_3$ serve as auxiliary invariants to prove $@inv1\_4$. We define the carrier set E to present the atrial and ventricle events. Three constants vs, vp and us are defined to present the VS event, VP event and unknown event respectively. In the machine, the variable s is used to present the latest event that has been observed, which is set to unknown initially. Either VS_Trig and VP_Trig could occur at the beginning. The timing property (7.1a) guarantees that VP_Resp event occurs within $[URI, LRI]$ time interval. In case of VP_Trig occurring, the timing property (7.1b) shows that a VS_Abt event could interrupt the timing properties between VP_Trig and VP_Resp events, but VP_Resp would occur within $[URI, LRI]$ after the VS_Abt event based on the time response pattern.

$$Abt(VP\_Trig, VP\_Resp, VS\_Abt, URI, LRI) \tag{7.1a}$$

$$Resp(\{VS\_Trig, VS\_Abt\}, VP\_Resp, URI, LRI) \tag{7.1b}$$

```
@inv1_1 s=vp⇒clk− t_vp_c≤ LRI
@inv1_2 s=vs⇒clk− t_vs≤ LRI
@inv1_3 s=us⇒clk≤ LRI
@inv1_4 (t_vp_p<t_vp_c⇒  t_vp_c−t_vp_p≥ URI ∧t_vp_c−t_vp_p≤LRI) ∨ (t_vs<t_vp_c⇒t_vp_c−t_vs≥
      URI ∧t_vp_c−t_vs≤LRI)
```

```
event VS_Trig extends XS
where
  @grd2 s=us
then
  @act4 t_vs:= clk
  @act5 s:= vs
end

event VS_Abt extends XS
where
  @grd2 s=vp ∨ s=vs
then
  @act4 t_vs:= clk
  @act5 s:= vs
end

event tick extends tick
where
  @grd2 s=vp⇒clk+1− t_vp_c≤LRI
  @grd3 s=vs⇒clk+1− t_vs≤LRI
  @grd4 s=us⇒clk+1≤LRI
end
```

```
event VP_Resp
where
  @grd1 s=vp ⇒ clk≥ t_vp_c+URI
  @grd2 s=vs ⇒ clk≥ t_vs+URI
  @grd3 s=vp ∨ s=vs
then
  @act1 t_vp_c:= clk
  @act2 t_vp_p:= t_vp_c
  @act3 s:= vp
end

event VP_Trig
where
  @grd1 s=us
then
  @act1 t_vp_c:= clk
  @act2 t_vp_p:= clk
  @act3 s:= vp
end
```

FIGURE 7.3: Model URI and LRI with Abort Pattern and Response Pattern

7.2.3 AEI and AVI

The AVI is used to maintain the appropriate delay between the atrial events (AP and AS) and the ventricular pacing (VP). If no VS occurred between AP and VP, then the pacemaker should deliver VP after the AVI has passed. In the second refinement, we introduce the AP and AS to model the real-time properties AEI and AVI. The AEI and AVI are specified with time intervals $[AEI\_min, AEI\_max]$ and $[AVI\_min, AVI\_max]$ respectively. The timing requirements require that when VP occurs, AP should occur within $[AEI\_min, AEI\_max]$ time interval provided the VS event did not occur. After AP occurs, the next VP should occur within the $[AVI\_min, AVI\_max]$ time interval provided the absence of VS event. In M_1, we use the abort pattern to model the interrupt event VS between two VP events. In the refined machine, VS still serves as the abort event between two VP events. We introduce the intermediate event AP_Resp and use the intermediate pattern to model AEI and AVI timing properties. A new constant ap is added to the context to denote the occurrence of AP event. A new variable s_avi is used to present the latest event that has been observed in the refined model with new timing properties. The timing property (7.1a) is refined to timing property (7.2a) with the intermediate event AP_Resp. The variable s_avi is used to set the guards of AP_Resp and VP_Resp based on Condition 2 and Condition 3 in Definition 6.3. Timing property (7.1b) is refined to timing properties (7.2b) and (7.2c) with the sequential refinement pattern. AS_Abt event is used as the interrupt event to interrupt the AEI time interval. The timing property (7.2b) is used to maintain synchrony between the atria and the ventricles. In Figure 7.4, the variable t_ap denotes the time at which AP_Resp occurs. @$inv2\_2$ shows that the time interval between t_xs_c and t_ap is bounded by AEI_min and AEI_max. @$inv2\_4$ shows that the time interval between t_ap and t_xs_c is bounded by AVI_min and AVI_max. @$inv2\_1$ and @$inv2\_3$ serves as auxiliary invariants. @$inv2\_6$ shows that provided AS occurs before AP, and then the pacemaker should deliver VP within AVI_max. In the refinement step, the sum of AEI and AVI specifies the time interval between two VP events. Thus $AEI\_min + AVI\_min \geq URI$ and $AEI\_max + AVI\_max \leq LRI$. Based on this axiom, we replace @$grd2$ of $tick$ event in M_1 with @$grd5$-@$grd8$ in M_2.

$$Interm(VP\_Trig, VP\_Resp, AP\_Resp, URI, LRI) \tag{7.2a}$$

$$Resp(\{AP\_Resp, AS\_Abt\}, VP\_Resp, AVI\_min, AVI\_max) \tag{7.2b}$$

$$Abt(VS\_Trig, AP\_Resp, AS\_Abt, AEI\_min, AEI\_max) \tag{7.2c}$$

7.2.4 VRP and PVARP

The VRP timing property is defined as the period during which the pacemaker is insensitive to incoming signals, which follows the ventricular event to filter out the noises in the ventricular channel to disable a new LRI interval. The $PVARP$ timing property follows the ventricular event to eliminates the sensing of retrograde P waves from ventriculoatrial conduction (Barold et al., 2010). In this case, we refine the VS_Abt, VP_Resp with the delay time VRP, and AP_Resp with the delay time $PVARP$. Figure 7.5 shows the formalism that extends the model with VRP and $PVARP$ timing properties. @$inv3\_1$ guarantees that provided AP occurs after VP, and the delay should be larger than PVARP. @$inv3\_2$ denotes that provided either VP or VS occurs after VP, the delay should be larger than VRP and any ventricular events within VRP should be ignored.

7.2.5 sAVI and pAVI

The AVI timing property can be further classified based on the initiating event. The sensed AVI initiated by AS could be programmed to a shorter value than the paced AVI initiated by AP to shorten the TARP during atrial sensing (Barold et al., 2010). In this refinement, we introduce two constants $tsAVI \in [AVI\_min, AVI\_max]$ and

```
axioms
@axm1 AEI_max+AVI_max≤ LRI
@axm2 AEI_min+AVI_min≥ URI
invariants
@inv2_1 s_avi=vp  ⇒ clk−t_vp_c≤  AEI_max
@inv2_2 t_vp_c<t_ap⇒t_ap−t_vp_c≥   AEI_min ∧ t_ap−t_vp_c≤AEI_max
@inv2_3 s_avi=ap⇒clk−t_ap≤AVI_max
@inv2_4 t_ap<t_vp_c⇒t_vp_c−t_ap≥ AVI_min  ∧  t_vp_c−t_ap≤AVI_max
@inv2_5 s_avi=as⇒clk−t_as ≤AVI_max
@inv2_6 s_avi=as∧t_as<t_vp_c⇒ t_vp_c−t_as≤AVI_max
```

```
event VS_Trig extends VS_Trig
then
  @act6 s_avi:= vs
end

event VS_Abt extends VS_Abt
then
  @act6 s_avi:= vs
end

event VP_Trig extends VP_Trig
where
  @grd2 s≠as⇒clk≥ t_ap+AVI_min
then
  @act4 s_avi:= vp
  @act5 t_ap:= clk
end

event VP_Resp extends VP_Resp
where
  @grd4 s_avi=ap
  @grd5 s≠as⇒clk≥ t_ap+AVI_min
then
  @act4 s_avi:= vp
end
```

```
event AS_Abt extends XS
where
  @grd2 s_avi=vp ∨ s_avi=vs
  @grd3 clk≤ t_vp_c+AEI_max
then
  @act3 t_as:= clk
  @act4 s_avi:=  as
end

event AP_Resp
where
  @grd1 s_avi=vp
  @grd3 clk≥ t_vp_c+ AEI_min
then
  @act1 t_ap:= clk
  @act2 s_avi:= ap
end

event tick refines tick
where
  @grd1 clk+1−t_xs_c≤  h_max
  @grd3 s=vs⇒clk+1− t_vs≤ LRI
  @grd4 s=us⇒clk+1≤ LRI
  @grd5 s_avi=vp ⇒ clk+1−t_vp_c≤  AEI_max
  @grd6 s_avi=us⇒clk+1−t_vp_c≤  AEI_max
  @grd7 s_avi=ap⇒clk+1−t_ap≤AVI_max
  @grd8 s_avi=as⇒clk+1−t_as ≤AVI_max
then
  @act1 clk:= clk+1
end
```

FIGURE 7.4: Model AEI and AVI with Intermediate Pattern and Response Patterns

```
@inv3_1 t_vp_c<t_ap⇒t_ap−t_vp_c≥  PVARP
@inv3_2 (t_vp_p<t_vp_c⇒  t_vp_c−t_vp_p≥ VRP ) ∨ (t_vs<t_vp_c⇒t_vp_c−t_vs≥ VRP )
```

```
event VS_Abt extends VS_Abt
where
  @grd3 clk≥  t_vp_c+VRP
end

event AP_Resp refines AP_Resp
where
  @grd1 s_avi=vp
  @grd3 clk≥ t_vp_c+ PVARP
then
  @act1 t_ap:= clk
  @act2 s_avi:= ap
end
```

```
event VP_Resp refines VP_Resp
where
  @grd1 s=vp ⇒ clk≥ t_vp_c+VRP
  @grd2 s=vs ⇒ clk≥ t_vs+VRP
  @grd3 s=vp ∨ s=vs
  @grd4 s_avi=ap
  @grd5 s≠as⇒clk≥ t_ap+AVI_min
then
  @act1 t_vp_c:= clk
  @act2 t_vp_p:= t_vp_c
  @act3 s:= vp
  @act4 s_avi:= vp
end
```

FIGURE 7.5: Extend Model with VRP and PVARP timing properties

$tpAVI \in [AVI\_min, AVI\_max]$ to represent $sAVI$ and $pAVI$ respectively. The timing property (7.2b) is refined to timing properties (7.3a) and (7.3b) with the alternative refinement pattern.

Figure 7.6 shows the model that refines AVI with sAVI and pAVI. We use $tpAVI$ and $tsAVI$ to replace AVI_max in @$grd7$ and @$grd8$ of $tick$ event. @$inv4\_2$ and @$inv4\_4$ are used to show timing intervals of (AP, VP) and (AS, VP). @$inv4\_1$ and @$inv4\_3$ serves as auxiliary invariants for @$inv4\_2$ and @$inv4\_4$ respectively.

$$Resp(AP\_Resp, VP\_Resp, AVI\_min, tpAVI) \tag{7.3a}$$
$$Resp(AS\_Abt, VP\_Resp, AVI\_min, tsAVI) \tag{7.3b}$$

```
@inv4_1 s_avi=ap⇒clk−t_ap≤tpAVI
@inv4_2 t_ap<t_vp_c⇒t_vp_c−t_ap≥ AVI_min  ∧  t_vp_c−t_ap≤tpAVI
@inv4_3 s_avi=as⇒clk−t_as ≤tsAVI
@inv4_4 s_avi=as∧t_as<t_vp_c⇒ t_vp_c−t_as≤tsAVI

event tick refines tick
where
  @grd7 s_avi=ap⇒clk+1−t_ap≤tpAVI
  @grd8 s_avi=as⇒clk+1−t_as ≤tsAVI
then
  @act1 clk:= clk+1
end
```

FIGURE 7.6: Extend Model with sAVI and pAVI timing properties

7.2.6 VSP

In real-world cases, oversensing in pacemakers would cause underpacing as the pacemakers assume the heart is behaving correctly. Crosstalk is one of the oversensing cases when the stimulus in one chamber is sensed in the other chamber when the pacemaker is operating under DDD mode. Increasing the sensing threshold of the ventricular channel could prevent false sensing (Jiang and Mangharam, 2011). But VSP is usually used in pacemakers to prevent crosstalk. In the fifth refinement, we introduce the VSP time interval into the model to prevent crosstalk. The AP event would trigger the VP event during the VSP time interval and produce a typical abbreviation of the paced AV interval, and no VS event could be triggered in between. We refine the $pAVI$ timing property to the VSP time interval, which we define as $[VSP\_min, VSP\_max]$. After AP occurs, VP should occur within the $[VSP\_min, VSP\_max]$ time interval.

Figure 7.7 shows the model that refines $pAVI$ with $[VSP\_min, VSP\_max]$. $@inv5\_2$ verifies the VSP time interval and $@inv5\_2$ serves as the auxiliary invariant to prove $@inv5\_2$. Also, we replace $grd7$ in the $tick$ event in M_4 with $@grd7$ in M_5 under the conditon that $VSP\_max < tpAVI$.

```
axioms
@axm5_1 VSP_min>0
@axm5_2 VSP_max<tpAVI

invariants
@inv5_1 s_avi=ap⇒clk−t_ap≤ VSP_max
@inv5_2 s_avi=ap ∧ t_ap<t_vp_c ⇒t_vp_c−t_ap≥ VSP_min ∧  t_vp_c−t_ap ≤ VSP_max

event VP_Resp extends VP_Resp
  where
    @grd7 s_avi=ap⇒clk≥ t_ap+VSP_min
end

event tick refines tick
where
  @grd1 clk+1−t_xs_c≤  h_max
  @grd3 s=vs⇒clk+1− t_vs≤ LRI
  @grd4 s=us⇒clk+1≤ LRI
  @grd5 s_avi=vp ⇒ clk+1−t_vp_c≤   AEI_max
  @grd6 s_avi=us⇒clk+1−t_vp_c≤   AEI_max
  @grd7 s_avi=ap⇒clk+1−t_ap≤ VSP_max
  @grd8 s_avi=as⇒clk+1−t_as ≤ tsAVI
then
  @act1 clk:= clk+1
end
```

FIGURE 7.7: Extend Model with VSP Time Interval to Avoid Crosstalk

Table 7.1: Proof Statistics of Pacemaker Case Study

Machine	Generated PO	Automatically Proved	%
M0	13	13	100
M1	35	35	100
M2	58	58	100
M3	12	12	100
M4	29	29	100
M5	16	16	100

7.2.7 Proof Statistics

Table 9.1 shows the proof statistics of the model. We adopt a stepwise approach to model the complicated timing cycles of pacemakers with five levels. Results show that the proof obligations are discharged automatically by using the patterns that encode real-time properties in Event-B models.

7.3 Evaluation and Conclusion

To date, many studies have attempted to model and verify the control algorithms for pacemakers. Jiang et al. (2011) used timed automata (Alur and Dill, 1994) to develop the random heart model and the pacemaker model to reason about the correction algorithms for the endless loop tachycardia (ELT) and pacemaker mode-switch clinical cases. The model checking tool UPPAAL (Larsen et al., 1997) has been used to reveal safety violations in the system specification. Méry et al. (2011) developed a single electrode pacemaker system with the Event-B formalization and implemented the different operating modes of pacemakers in stepwise refinement. An action-reaction pattern was used to model the causal order between events. Time constraints were put between action and reaction events to model real-time properties. Based on the action-reaction pattern proposed by Cansell et al. (2007), Poppleton and Rezazadeh (2012) developed the pacemaker model under AAI, VVI and VDD mode. Sulskus et al. (2016) presented a template-based timing constraint modeling scheme that modeled the time interval in Event-B models and applied the scheme in the pacemaker case study. Their work mainly focused on the finite-state space model. Model checking was used to verify the correctness of the system. However, our approach modeled the timed system with unbounded clock variable and uses theorem proving to verify the consistency of timing cycles.

Based on work that provided formal semantics of trigger-response properties in Chapter 4, we made new contributions by providing four generic real-time specification patterns and two refinement patterns that can be used to bridge the gap between informal requirements and formal specifications. The new patterns not only allow modeling of timing properties between trigger and response events, but also allow modeling of the interrupt and reoccurrence behavior in real cases. Templates were provided to show how to apply the patterns syntactically in Event-B models. We showed that the complicated timing cycles in pacemakers could be developed in a stepwise manner by applying different real-time specification patterns and refinement patterns. The proof results of the model also showed that the usage of patterns could help to discharge the Event-B proof obligations automatically in the Rodin tool.

Chapter 8

Formalization of Parameterized Timing Properties

In Chapter 6, we presented the formalization that extends Event-B models with real-time trigger-response properties. However, the formalization does not distinguish timing properties for different system design phases and cannot show the communication and competition between concurrent tasks in concurrent or distributed systems. In this chapter, we present the formalization of parameterized timing properties in Event-B models to model end-to-end timing properties. We distinguish end-to-end timing properties and scheduler-based timing properties from different system design phases. End-to-end timing properties are defined as high-level timing properties from the system requirement specification phase, which place discrete time properties on individual tasks. However, these end-to-end timing properties cannot describe the concurrent behavior of tasks precisely. In real-time systems, there are always several tasks running concurrently. High-level time constraints for each task cannot guarantee the timing behavior of the whole system. To model the behavior of these concurrent tasks, we defined scheduler-based timing properties as concrete timing properties for the system design phase, which place discrete time constraints on the scheduler which schedules the concurrent tasks. And we use unparameterised timing properties to model these scheduler-based timing properties. In this chapter, we first present a formal definition of parameterized timing properties together with the primary and auxiliary invariants required to prove that an Event-B model satisfies specific timing properties. Then we propose a refinement pattern for generating auxiliary invariants to replace parameterized timing properties to unparameterized timing properties

8.1 Parameterized Real-Time Trigger-Response Properties

In Event-B models, parameters of events can be used to treat the concurrency of the system (Butler, 2009). An Event-B machine first executes the initialization, followed by nondeterministically executing some enabled event. When modeling a concurrent system, instead of having separate atomic events for each task of the concurrent system, parameterized events can be used to model the atomic steps of each task. Additionally, in concurrent systems, parameterized trigger-response timing properties can be used to specify the timing properties of each task.

Given an Event-B machine M with event labels E, which contains a trigger-response pair (T, R), we extend our definition for the real-time trigger-response properties to parameterized trigger-response properties as in Definition 8.1. In this definition, we use X to denote the parameter value set. Assume that event e has a parameter p, $e.x$ is defined as the semantic label used to represent occurrence of event e with parameter p instantiated with value x. The representation of a parameterized event e, with parameter p and operating on state variable v is represented by guard predicate $G_e(p, v)$ and before-after predicate $S_e(p, v, v')$. The semantic labels $T.x$ and $R.x$ are used to denote the occurrence of trigger events T or response events R with parameter p instantiated with value x, which are formally presented as (8.1a) and (8.1b).

$$T.x \triangleq \textbf{where} \quad G_T(x, v) \quad \textbf{then} \quad S_T(x, v, v') \tag{8.1a}$$

$$R.x \triangleq \textbf{where} \quad G_R(x, v) \quad \textbf{then} \quad S_R(x, v, v') \tag{8.1b}$$

Definition 8.1 (Parameterized Trigger-Response Ordering Property)**.** Given an Event-B machine M with events E and invariants $I(v)$, a parameterized trigger-response pair has the form (T, R, X) where $T \subseteq E$ are trigger events with a parameter p, $R \subseteq E$ are response events with a parameter p, and $T \cap R = \varnothing$. M satisfies sequential ordering for (T, R, X) provided the following conditions hold.

1. $G_R(p, v) \Rightarrow \neg G_T(p, v)$
2. $I(v) \wedge G_t(p, v) \wedge S_t(p, v, v') \Rightarrow \neg G_t(p, v') \wedge G_R(p, v')$
3. $e \in E \setminus (T \cup R) \wedge I(v) \wedge G_e(v) \wedge S_e(v, v') \wedge G_R(p, v) \Rightarrow G_R(p, v')$
4. $I(v) \wedge G_r(p, v) \wedge S_r(p, v, v') \Rightarrow \neg G_r(p, v')$

In Definition 8.1, we use $G_T(p, v)$ to denote the disjunction of all $G_t(p, v)$ with $t \in T$ and $G_R(p, v)$ to denote the disjunction of all $G_r(p, v)$ with $r \in R$. When extending Event-B models with timing properties, the five conditions of Definition 8.1 are required to guarantee the sequential order of parameterized trigger-response pairs. Condition 1 specifies that when a trigger event is enabled, the response events must be disabled.

Condition 2 requires that once a trigger event $t.x$ occurs, it disables itself and enables some response event $r.x$. Condition 3 requires that each $e \in E \setminus (T \cup R)$ preserves the predicate $G_R(p, v)$. Condition 4 requires that each response event r disables itself.

Similar to Definition 3.10, we define parameterized real-time trigger-response properties in Definition 8.2. w and d are total functions from the parameter value set X to the natural number set $\mathbb{N}$, which define the delay and deadline for each specific parameter instance.

Definition 8.2 (Parameterized Real-Time Trigger-Response Property)**.** A parameterized real-time trigger-response property has the form:

$$\textbf{all} \text{ x } \textbf{in } X \ \textbf{Resp}(T.x, R.x, w(x), d(x)) \tag{8.2}$$

where delay $w \in X \rightarrow \mathbb{N}$ and deadline $d \in X \rightarrow \mathbb{N}$.

Similar to behaviors of a machine with unparameterized timing properties, the behaviors of a model with parameterized real-time trigger-response property in (8.2) satisfies two properties: 1) the number of *Tick* events between each trigger event $T.x$ and its corresponding response event $R.x$ is bounded by the delay time $w(x)$ and deadline time $d(x)$; 2) no two occurrences of $T.x$ are allowed without an occurrence of $R.x$ in between. In Event-B models, we construct Inv 1 to capture the fact that each unique trigger-response pair $(T.x, R.x)$ is bounded by the deadline time $d(x)$. Inv 1 formalizes the safety property that when a response event occurs, the time between trigger and response event should be bounded by $d(x)$.

$$\forall x \cdot x \in X \wedge \tau_T(x) \leq \tau_R(x) \Rightarrow \tau_R(x) - \tau_T(x) \leq d(x) \tag{Inv 1}$$

Figure 8.1 shows the formalization we use that extends the untimed machine with (T, R, X) to timed machine with parameterized timing properties in (8.2) for each trigger-response pair $(t.x, r.x)$ where $t \in T$ and $r \in R$. The formalism is similar to the one used to extend the machine with unparameterized timing properties. The timestamp variable $\tau_t \in X \rightarrow \mathbb{N}$ and $\tau_r \in X \rightarrow \mathbb{N}$ are set by the before-after predicate in the trigger and response events respectively. Additional constraints relating to $\tau_t(x)$ and $\tau_r(x)$ are imposed on the response event R and *Tick* event. Here $G_{Tick}(v)$ in equation (8.3) denotes the guard of *Tick* event. In the $G_t(p, v)$ and $G_r(p, v)$ predicate, we substitute the parameter p with x.

$$G_{Tick}(v) \triangleq \forall x \cdot x \in X \wedge G_R(x, v) \Rightarrow \mathit{clk} + 1 \leq \tau_T(x) + d(x) \tag{8.3}$$

With the formalization shown in Figure 8.1, the response is constrained by the guard $\mathit{clk} \geq \tau_t(x) + w(x)$, which guarantees that response event $R.x$ can occur only after the

```
event t
any p
where
  $G_t(p, v)$
then
  v:= $S_t(p, v, v')$
  $\tau_t(p)$:= clk
end
```

```
event r
any p
where
  $G_r(p, v)$
  clk $\geq$ $\tau_t(p)$+w(p)
then
  v:= $S_r(p, v, v')$
  $\tau_r(p)$:= clk
end
```

```
event Tick
where
  ∀x·x∈ X ∧ $G_r(x, v)$ ⇒ clk+1
      ≤ $\tau_t(x)$+d(x)
then
  clk:= clk+1
end
```

FIGURE 8.1: Formalization that Extends the Machine with parameterized timing properties (8.2) for each trigger-response pair $(t.x, r.x)$ where $t \in T$ and $r \in R$

delay time $w(x)$ has passed. No additional gluing invariants are required for the delay constraint. Thus, in this section we focus on deadlines and not the delays. Inv 2 is an auxiliary invariant that can be used to prove Inv 1, which defines the state where the trigger event occurred while the response event has not occurred, and the time between the current time and the timestamp of the trigger event should also be bounded by $d(x)$. We first construct Lemma 8.3 to prove that Inv 2 is preserved by all the events of the machine with timing encoded.

$$\forall x \cdot x \in X \Rightarrow (G_R(x, v) \Rightarrow clk - \tau_T(x) \leq d(x)) \qquad \text{(Inv 2)}$$

Lemma 8.3. *Given the formalization in Figure 8.1 that extends the machine with* ***all*** *x* ***in*** *X* ***Resp***$(T.x, R.x, w(x), d(x))$*, Inv 2 is preserved by all the events in the extended machine.*

Proof. Given Inv 2, the only events that change the variables of the invariant are trigger events $t \in T$ and *Tick* event. Thus in the proof we mainly examine these two events.

For the trigger events, preservation of Inv 2 is represented as follows:

$$\frac{\begin{array}{c} H_0 : \forall x \cdot x \in X \Rightarrow (G_R(x, v) \Rightarrow clk - \tau_T(x) \leq d(x)) \\ H_1 : I(v) \wedge G_t(p, v) \wedge S_t(p, v, v') \end{array}}{G : \forall x \cdot x \in X \Rightarrow (G_R(x, v') \Rightarrow clk' - \tau'_T(x) \leq d(x))}$$

Assume $x \in X$ and $G_R(x, v')$, we have to prove $clk' - \tau'_T(x) \leq d(x)$.

$$
\begin{array}{ll}
 & clk' - \tau'_T(x) \leq d(x) \\
 & \quad \langle \textbf{Case } \mathrm{x{=}p} \rangle \\
 & clk' - \tau'_T(p) \leq d(p) \\
\equiv & \quad \langle S_t(p, v, v') : clk' = clk; \tau'_T = (\tau_T \vartriangleleft\!\!- \{p \mapsto clk\}) \rangle \\
 & clk - (\tau_T \vartriangleleft\!\!- \{p \mapsto clk\})(p) \leq d(p) \\
\equiv & \quad \langle (\tau_T \vartriangleleft\!\!- \{p \mapsto clk\})(p) = clk \rangle \\
 & clk - clk \leq d(p) \\
\equiv & \quad \langle 0 \leq d(p) \rangle \\
 & \top \\
 & \quad \langle \textbf{Case } \ x \neq p \rangle \\
 & clk' - \tau'_T(x) \leq d(x) \\
\equiv & \quad \langle S_t(p, v, v') : clk' = clk; \tau'_T = (\tau_T \vartriangleleft\!\!- \{p \mapsto clk\}) \rangle \\
 & clk - \tau_T(x) \leq d(x) \\
\Leftarrow & \quad \langle H_0 \rangle \\
 & \top
\end{array}
$$

For the *Tick* event, the preservation of Inv 2 is represented by:

$$
\frac{\begin{array}{c} H_0 : \forall x \cdot x \in X \Rightarrow (G_R(x, v) \Rightarrow clk - \tau_T(x) \leq d(x)) \\ H_1 : \forall x \cdot x \in X \wedge G_R(x, v) \Rightarrow clk + 1 - \tau_T(x) \leq d(x) \\ H_2 : clk' = clk + 1 \end{array}}{G : \forall x \cdot x \in X \Rightarrow G_R(x, v') \Rightarrow clk' - \tau'_T(x) \leq d(x)}
$$

$$
\begin{array}{ll}
 & \forall x \cdot x \in X \Rightarrow G_R(x, v') \Rightarrow clk' - \tau'_T(x) \leq d(x) \\
\equiv & \quad \langle clk' = clk + 1; \tau'_T(p) = \tau_T(p) \rangle \\
 & \forall x \cdot x \in X \Rightarrow G_R(x, v') \Rightarrow clk + 1 - \tau_T(x) \leq d(x) \\
\equiv & \quad \langle H_1 \rangle \\
 & \top
\end{array}
$$

$\square$

In the formalization, *clk* is increased by the *Tick* event only. The difference between $\tau_T(p)$ and $\tau_R(p)$ stands for the number of *Tick* events between $T.x$ and $R.x$. When $\tau_T(p) \leq \tau_R(p)$, the response event $R.x$ must have occurred after the trigger event. Therefore, when Inv 1 is preserved by the model, the behavior of the model satisfies the properties of parameterized timing properties. Hence, we construct Theorem 8.4 to prove that Inv 1 is preserved by all the events in a machine, which shows that the behavior of the model satisfies the parameterized real-time trigger-response properties.

Theorem 8.4. *Given the formalization in Figure 8.1 that extends machine M with* ***all*** *x* ***in*** *X* ***Resp***$(T.x, R.x, w, d)$*, invariant Inv 1 is preserved by all the events in the extended machine.*

Proof. Given Inv 1, the only events that changes the τ_T variables and τ_R variables are t and r events where $t \in T$ and $r \in R$. Thus in the proof we only examine these two events.

For the trigger events, the preservation of Inv 1 is represented by:

$$\begin{array}{c} H_0 : \forall x \cdot x \in X \wedge \tau_T(x) \leq \tau_R(x) \Rightarrow \tau_R(x) - \tau_T(x) \leq d(x) \\ H_1 : I(v) \wedge G_t(p, v) \wedge S_t(p, v, v') \\ \hline G : \forall x \cdot x \in X \wedge \tau'_T(x) \leq \tau'_R(x) \Rightarrow \tau'_R(x) - \tau'_T(x) \leq d(x) \end{array}$$

$$\begin{array}{ll} & \forall x \cdot x \in X \wedge \tau'_T(x) \leq \tau'_R(x) \Rightarrow \tau'_R(x) - \tau'_T(x) \leq d(x) \\ \Leftarrow & \quad \langle x \text{ is not free in } H_0 \rangle \\ & \tau'_T(x) \leq \tau'_R(x) \Rightarrow \tau'_R(x) - \tau'_T(x) \leq d(x) \\ & \quad \langle \textbf{Case } \text{x=p} \rangle \\ & \tau'_T(p) \leq \tau'_R(p) \Rightarrow \tau'_R(p) - \tau'_T(p) \leq d(p) \\ \equiv & \quad \langle S_t(p, v, v') : \tau'_R = \tau_R ; \tau'_T = (\tau_T \mathbin{\lhd\!\!-} \{p \mapsto clk\}) \rangle \\ & \tau'_T(p) \leq \tau'_R(p) \Rightarrow \tau_R(p) - clk \leq d(p) \\ \equiv & \quad \langle 0 \leq d(p) \wedge \tau_R(p) \leq clk \rangle \\ & \tau'_T(p) \leq \tau'_R(p) \Rightarrow \top \equiv \top \\ & \quad \langle \textbf{Case } x \neq p \rangle \\ & \tau'_T(x) \leq \tau'_R(x) \Rightarrow \tau'_R(x) - \tau'_T(x) \leq d(x) \\ \equiv & \quad \langle S_t(p, v, v') : \tau'_R = \tau_R ; \tau'_T = (\tau_T \mathbin{\lhd\!\!-} \{p \mapsto clk\}) \rangle \\ & \tau_T(x) \leq \tau_R(x) \Rightarrow \tau_R(x) - \tau_T(x) \leq d(x) \\ \Leftarrow & \quad \langle H_0 \rangle \\ & \top \end{array}$$

For the response events, the preservation of Inv 1 is represented by:

$$\begin{array}{c} H_0 : \forall x \cdot x \in X \wedge \tau_T(x) \leq \tau_R(x) \Rightarrow \tau_R(x) - \tau_T(x) \leq d(x) \\ H_1 : I(v) \wedge G_R(p, v) \wedge S_r(p, v, v') \\ \hline G : \forall x \cdot x \in X \wedge \tau'_T(x) \leq \tau'_R(x) \Rightarrow \tau'_R(x) - \tau'_T(x) \leq d(x) \end{array}$$

$$
\begin{array}{ll}
 & \forall x \cdot x \in X \wedge \tau'_T(x) \leq \tau'_R(x) \Rightarrow \tau'_R(x) - \tau'_T(x) \leq d(x) \\
\Leftarrow & \quad \langle x \text{ is not free in } H_0 \rangle \\
 & \tau'_T(x) \leq \tau'_R(x) \Rightarrow \tau'_R(x) - \tau'_T(x) \leq d(x) \\
 & \quad \langle \textbf{Case} \quad x = p \rangle \\
 & \tau'_T(p) \leq \tau'_R(p) \Rightarrow \tau'_R(p) - \tau'_T(p) \leq d(p) \\
\equiv & \quad \langle S_t(p, v, v') : \tau'_T = \tau_T; \tau'_R = (\tau_R \lhd\!\!- \{p \mapsto clk\}) \rangle \\
 & \tau_T(p) \leq \tau_R(p) \Rightarrow clk - \tau_T(p) \leq d(p) \\
\Leftarrow & \quad \langle \text{strengthen predicate} \rangle \\
 & clk - \tau_T(p) \leq d(p) \\
\Leftarrow & \quad \langle \text{ Inv 2:} G_R(p, v) \Rightarrow clk - \tau_T(p) \leq d(p) \rangle \\
 & G_R(p, v) \\
\Leftarrow & \quad \langle H_1 \rangle \\
 & \top \\
 & \quad \langle \textbf{Case} \quad x \neq p \rangle \\
 & \tau'_T(x) \leq \tau'_R(x) \Rightarrow \tau'_R(x) - \tau'_T(x) \leq d(x) \\
\equiv & \quad \langle S_t(p, v, v') : \tau'_T = \tau_T; \tau'_R = (\tau_R \lhd\!\!- \{p \mapsto clk\}) \rangle \\
 & \tau_T(x) \leq \tau_R(x) \Rightarrow \tau_R(x) - \tau_T(x) \leq d(x) \\
\Leftarrow & \quad \langle H_0 \rangle \\
 & \top
\end{array}
$$

□

8.2 Replacing Parameterized Timing Property with Unparameterized Timing Properties

In concurrent computing, concurrent tasks are executed by interleaving the execution steps of each task, which models tasks in the outside world that happen concurrently. In real-time systems, scheduling is used to make sure that all tasks meet their deadlines (Alur, 2015). A scheduler is used to allocate the resource to a task for some time.

Parameterized timing properties describe time bounds over quantified trigger-response pairs. This method lacks an adequate representation of the conflicts of timing properties resulting from the competition between concurrent trigger-response pairs. Global schedulers are used to schedule the tasks to execute. We employ unparameterized timing properties $Resp(P, Q, 0, wt)$ to represent the timing properties of global schedulers, which can be used to replace end-to-end timing property in (8.4) with additional constraints. In this section, we present a pattern for replacing an end-to-end timing property with a collection of scheduler-based timing properties.

$$\textbf{all}\ \mathrm{x}\ \textbf{in}\ X\ \textbf{Resp}(T.x, R.x, 0, d(x)) \tag{8.4}$$

In the refinement, the timing properties at the abstract level could be replaced by other timing properties at the concrete level. One end-to-end timing property in (8.4) could be replaced by n scheduler-based timing properties presented in (8.5). Take HS system as an example, the abstract timing property (9.1b) is replaced by a set of concrete timing properties (9.2a) and (9.2b). We construct a pattern to replace the timing properties so that the refinement is preserved.

$$\begin{cases} \mathit{Resp}(P_1, Q_1, 0, wt_1) \\ \mathit{Resp}(P_2, Q_2, 0, wt_2) \\ \ldots \\ \mathit{Resp}(P_n, Q_n, 0, wt_n) \end{cases} \tag{8.5}$$

As presented in Figure 8.1, the guard of *Tick* event is determined by the timestamps of trigger events and the predicate G_R. Thus we use the relation between the timestamps of P and T and the relation between abstract event set R and concrete event sets $Q_1, Q_2, \ldots, Q_n$ to construct the gluing invariants that relate the end-to-end timing properties and scheduler-based timing properties.

When replacing the end-to-end timing properties with scheduler-based timing properties, the schedulers determine the waiting time of each task based on its scheduling policy. Thus, for each scheduler-based timing property $\mathit{Resp}(P_i, Q_i, 0, wt_i)$, where $i \in 1 \ldots n$, that replaces the end-to-end timing property **all** x **in** X $\mathit{Resp}(T.x, R.x, 0, d(x))$, we use a function $g_i(x)$ to denote the maximum waiting time of each waiting task x for other tasks to run under the specific scheduling policy. This $g_i(x)$ can be provided by modelers based on different real-time scheduling specifications, which can be used to relate the timestamps of P and T. We use $\tau_T(x)$ and $\tau_R(x)$ to represent the timestamps of events $T.x$ and $R.x$, and τ_P and τ_Q to represent the timestamps of events P and Q respectively. As shown in the formalism presented in Figure 8.1, the timestamps are updated when the corresponding trigger and response events occur. We require that $g_i(x)$ of each x is updated by the trigger event P simultaneously with τ_P. It is obvious that $g_i(x)$ should satisfy the condition presented in Equation (8.6), which requires that maximum waiting time of each task x should be less than its deadline $d(x)$.

$$\forall x \cdot x \in X \land G_R(x, v) \Rightarrow 0 \leq g_i(x) \leq d(x) \tag{8.6}$$

Figure 8.2 shows the time diagram of an example that replaces the parameterized timing property in (8.4) with a single unparameterized timing property $\mathit{Resp}(P, Q, 0, wt)$. Assume that tasks are being executed in the order $p_1 \rightarrow p_2 \rightarrow p_3$ and that the worst-case execution time is $wt \in \mathbb{N}$. p_3 has to wait for p_1 and p_2 to finish being executed before it is executed. In the refined model, some unparameterised event can be used to replace the parameterized events. For example, in the abstract model we use $T.x$ to denote the event that some task x wishes to run. $\tau_T(x)$ is used to denote the timestamp of each $T.x$ event. We also use $d(x)$ to denote the execution deadline for each task. In the refined model, we use event P to denote the event at which the scheduler allows some task to run and Q to denote the event that stops executing a running task. When τ_P is initially updated, the maximum waiting time for p_2 is wt and the maximum waiting time for p_3 is $2 * wt$. Thus $g(p_2) = wt$ and $g(p_3) = 2 * wt$. After p_1 finishes executing, τ_P is updated again by the scheduler. In this case p_2 does not need to wait and the maximum waiting time for p_3 is wt. After p_2 finishes executing, $g(p_3) = 0$.

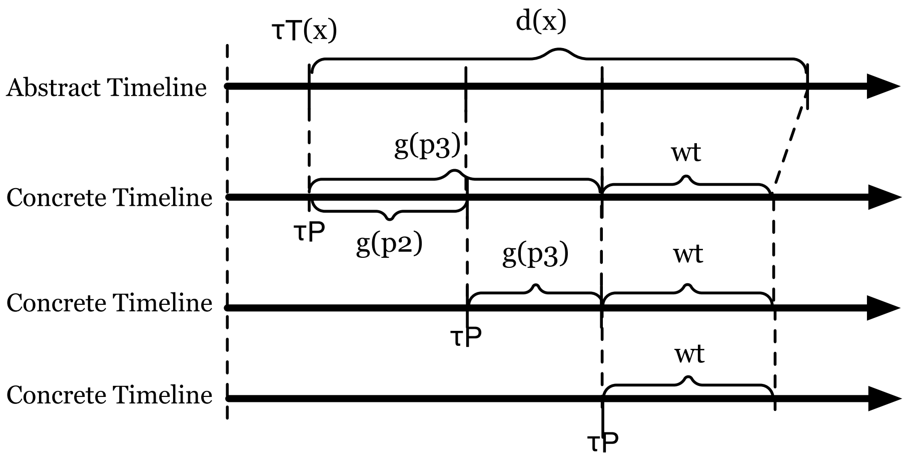

FIGURE 8.2: Time Diagram of Replacement of Parameterized Timing Properties to Unparameterized Timing Properties

Besides the relation between timestamps of trigger events, we also show the relation between the guards of the response events. For each task $x \in X$, we want to show the response events $R.x$ could be represented by some response event Q_i of the scheduler where $i \in 1 \ldots n$. Based on the above assumptions, we construct Theorem 8.5 to provide gluing invariants that relate the scheduler-based timing properties and end-to-end timing properties. We mainly show that the GRD proof obligation of *Tick* event is discharged by the additional conditions provided in Theorem 8.5.

Theorem 8.5. *Given a machine M with end-to-end timing property in (8.4) to be refined with a machine N with scheduler-based timing property presented in (8.5). Given that a function $g_i(x)$ that represents the waiting time of each task x under the specific scheduling policy is provided for each $\mathit{Resp}(P_i, Q_i, 0, wt_i)$ where $i \in 1 \ldots n$. The scheduler-based timing properties replace the end-to-end timing property when:*

1. *$H_{Q_i}(w) \Rightarrow (\tau_{P_i} + g_i(x) + wt_i \leq \tau_T(x) + d(x))$ is a valid invariant for each $Resp(P_i, Q_i, 0, wt_i)$ where $i \in 1 \ldots n$;*

2. *$\forall p \cdot G_R(p, v) \Rightarrow H_{Q_1}(w) \vee H_{Q_2}(w) \ldots \vee H_{Q_n}(w)$, where G_R is the guard of the response event of the end-to-end timing property, and H_{Q_i} is the guard of the response event of a timing property $Resp(P_i, Q_i, 0, wt_i)$ that replaces the abstract timing property.*

Proof. In this proof we want to show that the GRD proof obligation for the *Tick* event, formally presented as $I(v) \wedge J(v, w) \wedge H_{Tick}(w) \Rightarrow G_{Tick}(v)$.

$$\frac{\begin{array}{c} H_0 : G_R(p, v) \Rightarrow H_{Q_1}(w) \vee H_{Q_2}(w) \ldots \vee H_{Q_n}(w) \\ H_1 : (H_{Q_1}(w) \Rightarrow clk + 1 \leq \tau_{P_1} + wt_1) \wedge (H_{Q_1}(w) \Rightarrow (\tau_{P_1} + g_1(x) + wt_1 \leq \tau_T(x) + d(x))) \\ H_2 : (H_{Q_2}(w) \Rightarrow clk + 1 \leq \tau_{P_2} + wt_2) \wedge (H_{Q_2}(w) \Rightarrow (\tau_{P_2} + g_2(x) + wt_2 \leq \tau_T(x) + d(x))) \\ \cdots \\ H_n : (H_{Q_n}(w) \Rightarrow clk + 1 \leq \tau_{P_n} + wt_n) \wedge (H_{Q_n}(w) \Rightarrow (\tau_{P_n} + g_n(x) + wt_n \leq \tau_T(x) + d(x))) \end{array}}{G : \forall x \cdot x \in X \Rightarrow (G_R(x, v) \Rightarrow clk + 1 \leq \tau_T(x) + d(x))}$$

Assume $x \in X$ and $G_R(x, v)$. We have to show $clk + 1 \leq \tau_T(x) + d(x)$

$$\begin{array}{ll} & clk + 1 \\ \leq & \quad \langle H_0 : \exists i \cdot H_{Q_i} \text{ with } H_i \rangle \\ & \tau_{P_i} + wt_i \\ \leq & \quad \langle g_i(x) \geq 0 \rangle \\ & \tau_{P_i} + g_i(x) + wt_i \\ \leq & \quad \langle H_0 : \exists i \cdot H_{Q_i} \text{ with } H_i \rangle \\ & \tau_T(x) + d(x) \end{array}$$

□

8.3 Conclusion

Based on a trigger-response approach to modeling timing properties in Event-B, we presented the syntax of parameterized real-time trigger-response properties with Event-B formalization and proofs. Rules and proofs were provided to replace parameterized timing properties with unparameterized timing properties. To distinguish timing properties from the perspective of different system design phases, we defined parameterized timing properties that place discrete-time constraints on individual tasks as end-to-end timing properties, which describe high-level timing properties from the system requirement specification phase. These end-to-end timing properties cannot precisely describe the concurrent behavior of tasks. In real-time systems, schedulers are used to schedule concurrent tasks. To model the behavior of these concurrent tasks, we defined scheduler-based timing properties with unparameterized timing properties as concrete timing properties for the system design phase, which places discrete-time constraints on the scheduler that schedules the concurrent tasks. In Chapter 9, we used the rule that replaces the end-to-end timing properties with scheduler-based timing properties to model a two-level hierarchical scheduling system.

Chapter 9

Hierarchical Real-time Scheduling System

In Chapter 8, we present the syntax of parameterized real-time trigger-response properties with Event-B formalization and proofs. Rules and proofs are provided to replace parameterized timing properties with unparameterized timing properties. Based on the formalization and replacement pattern, we formalize a two-level hierarchical scheduling system that allows compositional scheduling policies in this chapter. Based on the replacement pattern, we develop a nondeterministic scheduling framework to replace end-to-end timing properties with scheduler-based timing properties. Then, this framework is refined to two alternative scheduling policies, namely, first-in-first-out (FIFO) and deferrable priority-based (DPB) scheduler with an aging technique. This chapter only introduces time-division multiplexing (TDM) as the global scheduler but also combines the two scheduling policies into one refinement to show that the local schedulers are compatible with different scheduling policies.

9.1 Hierarchical Scheduling System

With the emerging trend in real-time systems toward implementing functionalities in different levels on a shared platform, hierarchical scheduling (HS) systems are designed to use compatible schedulers to allocate CPU time so that all real-time applications meet their deadlines (Stankovic et al., 1998). In concurrent computing, concurrent tasks are executed by interleaving the execution steps of each task, which models tasks in the outside world that happen concurrently. When there are more tasks than processors, time slicing is usually used to simulate concurrency. In real-time systems, scheduling is used to make sure that all tasks meet their deadlines (Alur, 2015). A scheduler is used to allocate the resource between tasks with different execution time. Shared memory model is one of the common models for concurrent systems. In this section, we mainly show how to use Theorem 8.5 to replace parameterized timing properties with unparameterized timing properties. We use a two-level HS system to illustrate the replacement from end-to-end timing properties to scheduler-based timing properties under two assumptions. We first assume that the two-level hierarchical scheduling system only has one processor to run the tasks concurrently. We also assume that only one critical section is used for local resource sharing and there is no global resource sharing. Figure 9.1 shows a two-level HS that composes existing applications with different timing characteristics by using time-division multiplexing (TDM) as the global scheduler. The global scheduler decides which application should proceed and for how long. Then, each application uses its local scheduler to select which task to execute next. The TDM scheduler partitions a period into several time slots and assigns each of them to a single application. Take Figure 9.1 as an example. The TDM scheduler partitions the period of $50s$ into five time slots with $10s$ for each application. Then each application uses its local scheduler to schedule the concurrent tasks that might have dependencies on each other. In this case study, we do not address the cases of global resource sharing in HS systems. Therefore tasks of the same application might have a shared code segment that accesses shared variables. We use critical sections to present the code segment, which have to be executed as an atomic action. Tasks of different applications do not share critical sections. Our main system requirements (SRs) are as follows:

SR-1 Tasks of different applications do not share critical sections.

SR-2 No more than one application can be in the same time slot at any time.

SR-3 No more than one task of the same application can be in its critical section at any time.

SR-4 Each application is assigned a time slot to run the tasks.

SR-5 If a task wishes to enter its critical section when the application is running, it will enter the critical section within a certain deadline.

Figure 9.2 describes our refinement strategy to formalize the two-level HS system. In the abstract model, we first introduce a TDM global scheduler for applications and tasks in applications. Then we replace the end-to-end timing properties with scheduler-based timing properties by using a nondeterministic scheduling framework. This framework is refined to two alternative scheduling policies, namely, first-in-first-out (FIFO) and deferrable priority-based (DPB) scheduler with an aging technique. The refinement strategy is presented as follows:

M_0 specifies the local resource sharing system.

M_1 introduces end-to-end timing properties for individual tasks and allocates applications with time slots.

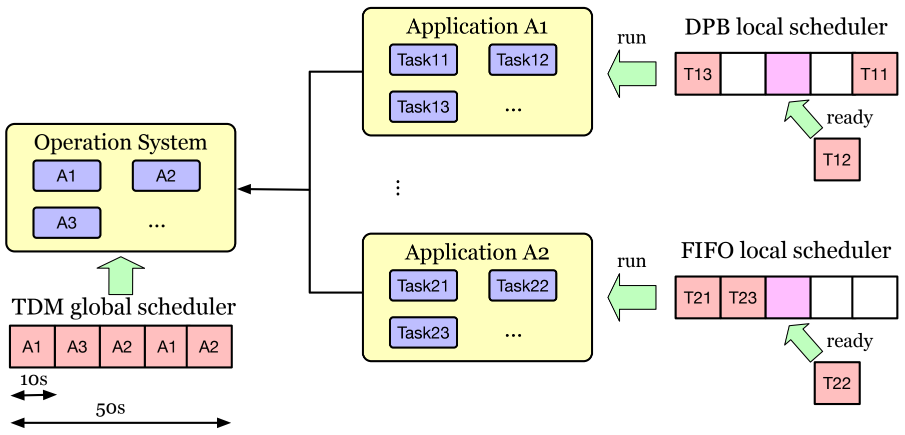

FIGURE 9.1: Two-level Hierarchical System

M_2 formalizes the sequence order of tasks with a nondeterministic queue-based scheduling framework

M_3 replaces the end-to-end timing property into scheduler based timing property with nondeterministic queue-based scheduling framework.

M_4 refines the nondeterministic queue-based scheduling framework into two alternative local schedulers.

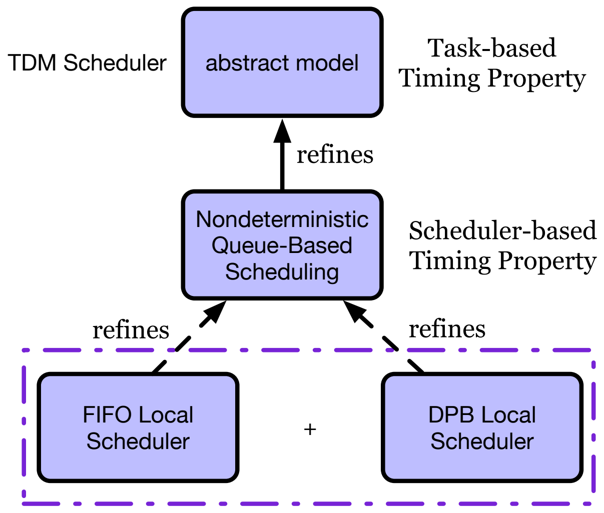
TDM Scheduler
abstract model
Task-based Timing Property
refines
Nondeterministic Queue-Based Scheduling
Scheduler-based Timing Property
refines
refines
FIFO Local Scheduler
+
DPB Local Scheduler

FIGURE 9.2: Refinement Strategy of Two-level Hierarchical System

In the next section, we explain these models and refinements in more detail and present part of our formalization.

9.2 Formalizing Hierarchical Scheduling with Timing Properties

9.2.1 Formalizing TDM and Local Resource Sharing with Mutual Exclusion

In the most abstract level M_0, a mutual exclusion model is proposed to address the dependencies within applications and tasks. The model guarantees that no two applications can be in the same time slot simultaneously and that no two tasks of the same application can be in the critical section simultaneously. We begin by defining the initial context in Figure 9.3. In the context, we define the carrier set *APPS* and *TASKS* of all applications and tasks in the HS system. We define *apps* as a total surjection from *TASKS* to *APPS* as a task cannot belong to two different applications. Additionally, the set of tasks of each application is finite, and its value is less than N.

```
sets APPS TASKS
constants apps N
axioms
   @axm0_1 apps∈ TASKS ↠ APPS
   @axm0_2 ∀a·a∈ APPS ⇒ finite(apps∼[{a}])
   @axm0_3 ∀a·a∈ APPS ⇒ card(apps∼[{a}])≤ N
   @axm0_4 N>0
end
```

FIGURE 9.3: Initial Context c0

In our initial model in Figure 9.4, we formalize the mutual exclusion model by using *app_run* and *task_run* variables. The variable *app_run* represents the set of applications that are currently running, whereas the variable *task_run* represents the set of tasks that are currently being executed. Variables *app_wait* and *task_wait* denote the set of applications and tasks that are ready to run. The invariants **inv0_1-inv0_4** formalize the relationships between the ready or waiting applications and tasks to the whole set. The invariants **inv0_5-inv0_6** guarantee that only one application and one task can be executed at any time. **inv0_7** guarantees that the tasks that are waiting or being executed belong to the application that is being executing.

In the abstract model, we use three events, namely, *ready*, *run* and *finish*, to model the behaviors of applications and tasks ready to run, execute and finish executing, respectively. As the mutual exclusion model is similar for applications and tasks, we take the events of applications as an example to illustrate the model. Figure 9.4 gives the abstract mutual exclusion model for applications. Here, we use quantified variable a to represent an application. The event $APP\_READY(a)$ models the point at which application a is ready to run. Event $APP\_RUN(a)$ models the point at which the application starts running, while event *APP_FINISH*(a) models the point at which the application finishes running. We use the same approach to model the behaviors of tasks of an application. Since the ready and running tasks should belong to the application that is being executed, we add an additional guard $apps(t) \in app\_run$ to the *TASK_READY* event to guarantee **inv0_7**. Take Figure 9.1 as an example. Applications A_1, A_2 and A_3 wish to run, and the TDM global scheduler assigns each application a time slot to run. When application A_2 is running, it executes different tasks with the DPB scheduling policy. For example, tasks T_{21}, T_{23} and T_{22} wish to be executed with the *TASK_READY* event. Then the *TASK_RUN*(T_{21}) event is executed since T_{21} is at the head of the queue. After T_{21} finishes, *TASK_RUN*(T_{23}) is executed.

```
invariants
@inv0_1 app_wait ⊆ APPS
@inv0_2 app_run ⊆ APPS
@inv0_3 task_wait ⊆ TASKS
@inv0_4 task_run ⊆ TASKS
@inv0_5 finite(app_run) ∧ card(app_run)≤ 1
@inv0_6 finite(task_run) ∧ card(task_run)≤ 1
@inv0_7 ∀t·t∈ (task_wait ∪ task_run)⇒apps(t)∈ app_run
```

```
event APP_READY
any a
where
  @grd1 a∈ APPS\app_wait
  @grd2 a∈ APPS\app_run
then
  @act1 app_wait:= app_wait∪{a}
end

event APP_RUN
any a
where
  @grd1 a∈ app_wait
  @grd2 app_run=∅
then
  @act1 app_wait:= app_wait\{a}
  @act2 app_run:= app_run ∪ {a}
end

event APP_FINISH
any a
where
  @grd1 a∈ app_run
  @grd2 task_wait=∅ ∧ task_run=∅
then
  @act1 app_run:= app_run\{a}
end
```

```
event TASK_READY
any t
where
  @grd1 t∈ TASKS\task_wait
  @grd2 t∈ TASKS\task_run
  @grd4 apps(t)∈ app_run
then
  @act1 task_wait:= task_wait∪{t}
end

event TASK_RUN
any t
where
  @grd1 t∈ task_wait
  @grd2 task_run=∅
then
  @act1 task_wait:= task_wait\{t}
  @act2 task_run:= task_run ∪ {t}
end

event TASK_FINISH
any t
where
  @grd1 t∈ task_run
then
  @act1 task_run:= task_run\{t}
end
```

FIGURE 9.4: Initial Model with TDM and Local Resource Sharing

Scheduling is used to allocate the processing time for concurrent tasks to maximize real-time performance (Alur, 2015). To guarantee that all the high-level timing requirements of individual tasks are satisfied by the system, we begin by formalizing high-level timing properties with the parameterized real-time trigger-response properties defined in Section 8.1. Then, we use different scheduling policies to replace the end-to-end timing properties with scheduler-based timing properties. Additional gluing invariants are provided based on the proposed refinement rules.

9.2.2 End-to-end Timing Properties

We define end-to-end timing properties as high-level timing properties to specify the time constraints of individual tasks in the HS system. Based on the abstract model that specifies no two tasks of the same application can be in the critical section simultaneously, we introduce parameterized real-time trigger-response properties to the model as end-to-end timing properties (9.1a) and (9.1b) in the first refinement. Timing property (9.1a) ensures that each application occupies the CPU resource for app_ddl. For each a in the $APPS$ set, the time between the occurrence of $APP\_READY.a$ and $APP\_RUN.a$ is bounded below by 0 and above by app_ddl. Timing property (9.1b) guarantees that if a task wishes to enter its critical section, it will enter the critical section within the specified timing property $task\_ddl$. Figure 9.5 shows the refinement with end-to-end timing property for each application. The timing property of each task can be modeled with the same pattern. We use @$axm1\_3$ to guarantee that all tasks of one application finish executing within the application execution time. $at(a)$ models the timestamp at which application a wishes to occupy CPU time. $ar(a)$ models the timestamp at which application a gets the CPU to run tasks. $tt(t)$ models the timestamp at which task t wishes to enter the critical section. $tr(t)$ models the timestamp task entering the critical section. @$inv1\_6$ and @$inv1\_8$ capture the end-to-end timing property based on parameterized real-time trigger-response property semantics. Since timed machine of Figure 9.5 is constructed according to the approaches of Section 8.1, from Theorem 8.4 we have that @$inv1\_6$ to @$inv1\_9$ are preserved.

$$\textbf{all a in } APPS \textbf{ Resp}(APP\_READY.a, APP\_RUN.a, 0, app\_ddl) \tag{9.1a}$$

$$\textbf{all t in } TASKS \textbf{ Resp}(TASK\_READY.t, TASK\_RUN.t, 0, task\_ddl) \tag{9.1b}$$

9.2.3 Replacing End-To-End Timing Property with Scheduler-Based Timing Properties

In the next refinement of the case study, we specify two scheduler-based timing properties with unparameterized timing properties (9.2a) and (9.2b). Property (9.2a) requires

```
constants app_ddl task_ddl
axioms
    @axm1_1 app_ddl>0
    @axm1_2 task_ddl>0
    @axm1_3 N ∗ task_ddl≤ app_ddl
invariants
    @inv1_6 ∀a·at(a)≤ ar(a)⇒ ar(a)−at(a)≤ app_ddl
    @inv1_7 ∀a·a∈ app_wait⇒clk−at(a)≤ app_ddl
    @inv1_8 ∀t·tt(t)≤ tr(t)⇒ tr(t)−tt(t)≤ task_ddl
    @inv1_9 ∀t·t∈ task_wait⇒clk−tt(t)≤ task_ddl
```

```
event APP_READY extends APP_READY
then
    @act2 at(a):= clk
end

event APP_RUN extends APP_RUN
    then
    @act3 ar(a):= clk
end
```

```
event TICK
where
    @grd1 ∀a·a∈ app_wait  ⇒ clk+1−at(a)≤
      app_ddl
    @grd2 ∀t·t∈ task_wait  ⇒ clk+1−tt(t)≤
      task_ddl
then
    @act1 clk:= clk+1
end
```

Figure 9.5: First Refinement with End-to-end Timing Properties

that when the system is idle, one of the requesting tasks will enter the critical section within *idletime*. Specifically, there are two cases that trigger the scheduling of the enter event: 1) a task wishes to enter, and both the queue and the critical section are empty, and 2) some task leaves the critical section, and there is some other task waiting in the queue. Observe here that events can act as timing triggers only under certain conditions; e.g., the *wish* event is only a timing trigger when the queue and critical section are empty. To address such conditional triggers, we split the event into separate refinements representing separate cases. We refine the *TASK_READY* event into a *TASK_READY_EMPTY* event, enabled when the first condition holds, and a *TASK_READY_NONEMPTY* event, enabled when the second condition holds. Similarly, we split the *TASK_FINISH* event into a *TASK_FINISH_NONEMPTY* event, enabled when the second condition holds, and a *TASK_FINISH_IDLE* event, enabled when the last task in the queue finish executing. The events *TASK_READY_EMPTY* and *TASK_FINISH_NONEMPTY* are therefore used as trigger events in (9.2a), whereas the response event *TASK_RUN* is the event modeling entering the critical section.

(9.2b) requires that once a task enters the critical section, it will leave the critical section within *runtime*. Therefore, the trigger event is the *TASK_RUN* event, whereas response events should correspond to leaving the critical section. As the latter is now captured by *two* events, there are two response events in (9.2b). Here, *TASK_FINISH_NONEMPTY* implies that when some tasks finish executing, others are still waiting in the queue. *TASK_FINISH_IDLE* denotes the situation in which the last task in the queue finishes executing.

$$\begin{cases} Resp(TASK\_READY\_EMPTY, TASK\_RUN, 0, idletime) \\ Resp(TASK\_FINISH\_NONEMPTY, TASK\_RUN, 0, idletime) \end{cases} \tag{9.2a}$$

$$\begin{cases} Resp(TASK\_RUN, TASK\_FINISH\_NONEMPTY, 0, runtime) \\ Resp(TASK\_RUN, TASK\_FINISH\_IDLE, 0, runtime) \end{cases} \tag{9.2b}$$

9.2.4 Nondeterministic Queue-Based Scheduling

In our HS system that replaces the end-to-end deadline constraint with scheduler-based deadline constraints, we propose a nondeterministic queue-based scheduling framework to address the scheduling order of the sequential execution of a set of events. In this framework, a queue is used to manage the ready tasks. Each task is formally assigned a position in the queue: $queue \in wait \rightarrowtail (0..N-1)$. When one task is ready, it is nondeterministically assigned a natural number that is not in the range of the queue. Only the task in the front of the queue $(min(ran(queue)))$ can get the resource to run. The dequeue operation will decrease the indexes of all other tasks in the queue by the index of the front task plus one $(min(ran(queue)) + 1)$ to guarantee that once a task is added to the queue, and it will eventually have the opportunity to run. In the second refinement, we use this nondeterministic queue-based scheduling framework to impose an order on the execution of the concurrent tasks. This refinement prevents a task from entering the critical section endlessly while also not allowing other tasks to enter the critical section. The second refinement is shown in Figure 9.6.

```
invariants
@inv2_1 queue∈ task_wait ↣  0..N−1
```

```
event TASK_READY extends TASK_READY
any i
where
  @grd5 i∈ 0..N−1
  @grd6 i∉ ran(queue)
then
  @act3 queue(t):= i
end
```

```
event TASK_RUN extends TASK_RUN
any j
where
  @grd3 queue≠∅
  @grd4 j∈ ran(queue)
  @grd5 j=min(ran(queue))
  @grd6 t=queue∼(j)
then
  @act4 queue:= (λ q·q∈ dom({t}⩤queue) |
     queue(q)−j−1)
end
```

FIGURE 9.6: Second Refinement with Nondeterministic Queue Based Scheduling

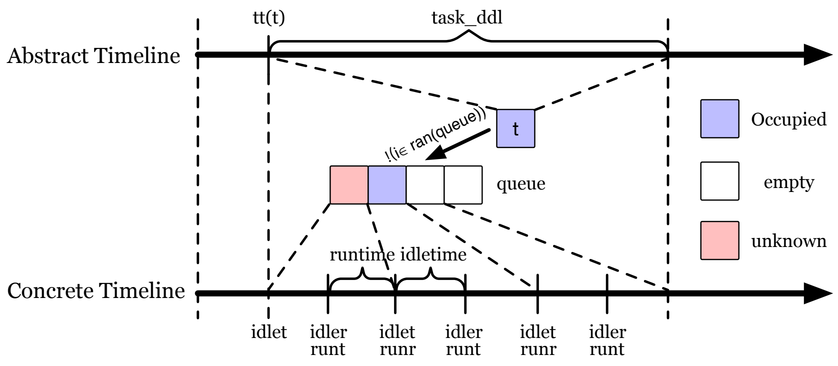

Figure 9.7: Time Diagram of Timing Properties' Refinement with the Scheduling Framework

Figure 9.7 shows the time diagram of the refinement with the scheduling framework. Assume that in the abstract machine, the trigger event of one task t occurs at timestamp $tt(t)$, and the deadline is $task\_ddl$. Additional gluing invariants are provided based on Theorem 8.5. In the refined machine, the trigger event $TASK\_READY\_EMPTY$ or $TASK\_FINISH\_NONEMPTY$ starts at timestamp $idlet$, and its deadline is $idletime$. The trigger event $TASK\_RUN$ starts at timestamp $runt$, and its deadline is $runtime$. The task t has to wait for all the tasks ahead of it in the queue to enter and leave the critical section. The total waiting time is proportional to its index in the queue, which is $queue(t)*(idletime+runtime)$. If the critical section is empty and the time that the last task leaves the critical section is $idlet$, then $g(t) = queue(t)*(idletime+runtime)$. There might be tasks that have waited $queue(t) * (idletime + runtime)$. Then, task t should enter the critical section within $idlet + queue(t) * (idletime + runtime) + idletime$. Given that the critical section is not empty, $g(t) = queue(t)*(idletime+runtime)$. Assume that the time that the last task enters the critical section is $runt$; then, task t should enter the critical section within $runt + queue(t) * (idletime + runtime) + (idletime + runtime)$. Based on Theorem 8.5, the sum of the refined sequential deadline should be less than the abstract deadline $tt(t) + task\_ddl$, which is shown in @$inv3\_9$ and $inv3\_10$ in Figure 9.8. @$inv3\_9$ and $inv3\_10$ present these two conditions as required gluing invariants. Assume that there are N tasks, of which the worst case is $N - 1$ tasks in the waiting list; thus, $max(queue(t)) = N - 1$. @$axm2\_3$ and @$axm2\_4$ present the required condition. Figure 9.8 shows the required axioms and invariants to replace the end-to-end deadlines to scheduler-based deadlines. @$inv3\_5$ and @$inv3\_7$ present the invariant for scheduler-based deadlines (9.2b), and @$inv3\_6$ and @$inv3\_8$ present the invariants for scheduler-based deadlines (9.2a).

```
constants idletime runtime
axioms
   @axm2_1 idletime>0
   @axm2_2 runtime>0
   @axm2_3 ((N−1)∗(idletime+runtime))+idletime≤ task_ddl
   @axm2_4 N∗(idletime+runtime)≤ task_ddl
invariants
@inv3_5 idler≥ idlet⇒ idler−idlet≤ idletime
@inv3_6 runr≥ runt⇒runr−runt≤ runtime
@inv3_7 queue≠∅ ∧ task_run=∅ ⇒ clk−idlet≤ idletime
@inv3_8 task_run≠ ∅⇒clk−runt≤ runtime
@inv3_9 ∀t·task_run=∅∧t∈ task_wait⇒idlet+(queue(t)∗(idletime+runtime))+idletime≤ tt(t)+
      task_ddl
@inv3_10 ∀t·task_run≠∅∧t∈ task_wait⇒runt+(queue(t)∗(idletime+runtime))+(idletime+runtime)
      ≤ tt(t)+task_ddl
```

```
TASK_READY_EMPTY
      extends TASK_READY
  where
     @grd7 task_wait=∅∧
      task_run=∅
  then
     @act4 idlet:= clk
end

TASK_READY_NONEMPTY
      extends TASK_READY
  where
     @grd7 task_wait≠∅ ∨
      task_run≠∅
end
```

```
TASK_RUN extends
      TASK_RUN
  then
   @act5 runt:= clk
   @act6 idler:= clk
end

TASK_FINISH_NONEMPTY
      extends TASK_FINISH
  where
     @grd2 queue≠∅
  then
     @act2 runr:= clk
     @act3 idlet:= clk
end
```

```
TASK_FINISH_IDLE extends
      TASK_FINISH
where
   @grd2 queue=∅
then
   @act2 runr:= clk
end
TICK refines TICK
where
   @grd1 ∀a·a∈ app_wait  ⇒
     clk+1−at(a)≤ app_ddl
   @grd2 task_run=∅ ∧
     task_wait≠∅ ⇒ clk+1−
     idlet≤ idletime
   @grd3 task_run≠∅⇒clk
     +1−runt≤ runtime
then
   @act1 clk:= clk+1
end
```

FIGURE 9.8: Replace Task-based Timing Properties with Scheduler-based Timing Properties

9.2.5 Two Implementations of Nondeterministic Queue-Based Scheduling

The nondeterministic queue-based scheduling framework is a general framework that nondeterministically assigns indexes to tasks. By applying additional rules to the assignment of these indexes, the queue-based scheduling framework can be refined to some scheduling policies such as the *FIFO* and *DPB* scheduling policies. In the model, we define *FIFO* and *DPB* as two scheduling policies of the *SCHEDULING* carrier set. The refinement shows that the two scheduling policies are compatible under the HS system. Each application has either the *FIFO* or *DPB* scheduling policy based on the *scheduling* constant. As shown in Figure 9.9, we define the constant *scheduling* as a total function from *APPS* to *Scheduling*. Both refinements refine the nondeterministic queue base scheduling framework by restricting the position of the ready task in the queue. Details are provided in the following sections.

9.2.5.1 First In First Out

FIFO is one of the scheduling policies that guarantees that the resources are assigned to each task in the order that they require the resource. The FIFO scheduling policy handles all tasks without priorities. The queue-based scheduling framework assigns each task with a corresponding natural number $k \in \mathbb{N}$, and the FIFO scheduling policy limits this natural number to the current size of the queue. Moreover, when the critical section is empty, the task that is in the front of the queue leaves the queue and enters the critical section. The indexes of all the other tasks in the queue are reduced by one.

The refinement from the scheduler-based model is shown in Figure 9.9. Assume that the application that is running is scheduling tasks with the *FIFO* policy. Initially, the queue is empty, and *qsize* is zero. Whenever some task is added to the queue, it is assigned the number of the queue size, and the queue size increases by one when the scheduling policy is *FIFO*. When the critical section is empty, the task in the front of queue $queue(0)$ is removed from the queue, and the indexes of all the other tasks in the queue are reduced by one. The queue size also decreases by one. In the nondeterministic queue-based scheduling policy, the guard for $TASK\_READY$ is $i \notin ran(queue)$. @$inv4\_1$ and @$inv4\_2$ are used to to prove that $i = qsize \Rightarrow i \notin ran(queue)$. $TASK\_READY\_NONEMPTY$ uses the same refinement strategy as $TASK\_READY\_EMPTY$.

9.2.5.2 Deferrable Priority Based Scheduling with Aging Technique

Fixed priority scheduling policies assign tasks with fixed priorities. In the model, we use $pindex \in TASKS \rightarrow 0..N-1$ to denote the queue position of tasks with different priorities. Tasks with higher priorities have lower indexes in the queue. The scheduler will select the tasks with higher priorities to access the system resources before those with lower priorities. However, there is a disadvantage of these scheduling policies: tasks with lower priorities may be starved when the tasks with higher priorities keep coming and jumping the queue. An aging technique is used to ensure that tasks with lower priorities are eventually executed. The general way to implement an aging technique is to increase the priorities of tasks with lower priorities while they are waiting in the ready queue. However, with the increasing priorities of some tasks, aging will allow tasks with lower priorities to occupy the positions of other tasks. In contrast, deferrable priority-based scheduling allows a task to be deferred with a random position after its assigned position when some other tasks occupy the position of that task.

To avoid the starving problem of tasks with lower priorities, we add a rule to priority-based scheduling: when some other task with lower priority occupies the position of

```
sets SCHDULING

constants FIFO DPB scheduling

axioms
  @axm3_2 partition(SCHDULING,{FIFO},{DPB})
  @axm3_3 scheduling∈ APPS → SCHDULING
invariants
  @inv4_1 qsize∈ 0..N
  @inv4_2 ∀i·i≥ qsize∧app_run⊆ scheduling∼[{FIFO}]⇒i∉ ran(queue)
```

```
TASK_READY_NONEMPTY_FIFO refines
     TASK_READY_NONEMPTY
any t i
where
    @grd1 t∈ TASKS\task_wait
    @grd2 t∈ TASKS\task_run
    @grd4 apps(t)∈ app_run
    @grd5 i=qsize
    @grd6 qsize≤ N−1
    @grd7 task_wait≠∅ ∨ task_run≠∅
    @grd8 app_run⊆ scheduling∼[{FIFO}]
then
    @act1 task_wait:= task_wait∪{t}
    @act2 tt(t):= clk
    @act3 queue(t):= i
    @act4 idlet:= clk
    @act5 qsize:= qsize+1
end
```

```
TASK_RUN_FIFO refines TASK_RUN
any t j
where
    @grd1 t∈ task_wait
    @grd2 task_run=∅
    @grd3 queue≠∅
    @grd4 j∈ ran(queue) ∧ j=0
    @grd5 t=queue∼(j)
    @grd6 app_run⊆ scheduling∼[{FIFO}]
then
    @act1 task_wait:= task_wait\{t}
    @act2 task_run:= task_run ∪ {t}
    @act3 tr(t):= clk
    @act4 queue:= (λ q·q∈ dom({t}⩤queue)|queue(q)−
      j−1)
    @act5 runt:= clk
    @act6 idler:= clk
    @act7 qsize:= qsize−1
end
```

FIGURE 9.9: First In First Out Scheduling Policy

some high-priority tasks, which means that the lower priority one has waited some time in the queue, the high-priority one is deferred by some higher random index. Specifically, the indexes of the tasks are decreasing by $min(ran(queue)) + 1$ when the task at the front queue, whose index is $min(ran(queue))$, leaves the queue and enters the critical section. The *enqueue* operation will assign the task its corresponding index in the queue. However, this operation would cause a conflict, as it will make some tasks occupy the spaces of other tasks. For example, task a's level is 3, and task b's level is 2. One task c is at the front of the queue. When c leaves the queue, the index of a is reduced to 2. When b wishes to enter the queue, its position is taken by a. Here, we choose the next available space available in the queue $i = min(k \mid k \in ran(pindex) \wedge k \notin ran(queue) \wedge k > pindex(t))$. When other tasks do not take the position, the task takes its assigned position $pindex(t)$. The dequeue operation is the same as the basic queue-based scheduling framework. Figure 9.10 shows the refinement from the scheduler-based model with a deferrable priority-based scheduling policy with aging technique.

```
constants pindex
@axm3_1 pindex∈ TASKS → 0..N−1

TASK_READY_NONEMPTY_DPB refines TASK_READY_NONEMPTY
any t i
where
    @grd1 t∈ TASKS\task_wait
    @grd2 t∈ TASKS\task_run
    @grd4 apps(t)∈ app_run
    @grd5 {k|k∈ ran(pindex)∧k∉ ran(queue)∧k≥ pindex(t)}≠∅
    @grd6 i=min({k|k∈ ran(pindex)∧k∉ ran(queue)∧k≥ pindex(t)})
    @grd7 task_wait≠∅ ∨ task_run≠∅
    @grd8 app_run⊆ scheduling∼[{DPB}]
then
    @act1 task_wait:= task_wait∪{t}
    @act2 tt(t):= clk
    @act3 queue(t):= i
end

TASK_RUN_DPB extends TASK_RUN
where
    @grd7 app_run⊆ scheduling∼[{DPB}]
end
```

FIGURE 9.10: Deferrable Priority Based Scheduling Policy with Aging Technique

TABLE 9.1: Proof Statistics of Two-Level Hierarchical Scheduling Case Study

Machine	Generated PO	Automatically Proved	Automatically Proved %
m0	14	12	85.7
m1	34	34	100
m2	6	5	83.3
m3	49	43	87.8
m4	27	27	100

9.2.6 Proof Statistics

Table 9.1 shows the proof statistics of the model. In $m3$, six proof obligations, all of which are related to @$inv3\_9$ and @$inv3\_10$, cannot be discharged automatically. As mentioned in Theorem 8.5, the modeler needs to provide the function $g(x)$ that denotes the maximum waiting time of each task x based on different scheduling policies. Thus, the modeler needs to verify that these two invariants are consistent with the whole model. Then, @$inv3\_9$ and @$inv3\_10$ can be used to discharge the *GRD* proof obligation of the *Tick* event. In $m4$, we provide @$inv4\_2$ to capture the property that given the *FIFO* scheduling policy, any i larger or equal to $qsize$ is not in the range of $queue$. @$inv4\_2$ helps with the proof obligations to refine the nondeterministic queue-based scheduling policy to the *FIFO* policy. Therefore, all proof obligations in $m4$ are discharged. The model and the proofs supporting this study are openly available from the University of Southampton repository at (Zhu et al., 2019a).

9.3 Conclusion

In this chapter, we formalized a two-level hierarchical scheduling system that refines the nondeterministic queue-based scheduling policies to two compatible local scheduling policies to illustrate the pattern that replaces end-to-end timing properties with scheduler-based timing properties. To model the behavior of these concurrent tasks, we defined scheduler-based timing properties with unparameterized timing properties as concrete timing properties for the system design phase, which places discrete-time constraints on the scheduler that schedules the concurrent tasks. To replace end-to-end timing properties with scheduler-based timing properties, we introduced a nondeterministic queue-based scheduling policy with some additional gluing invariants. We formalized a two-level hierarchical scheduling system that refines the nondeterministic queue-based scheduling policies to two compatible local scheduling policies to illustrate the pattern that replaces end-to-end timing properties with scheduler-based timing properties.

Chapter 10

Conclusions and Future Work

10.1 Main Contributions

To summarize, our work provided formal treatment of real-time properties in Event-B models from both the semantics perspective and syntax perspective.

In Chapter 3 and Chapter 4, we developed the semantics of real-time properties in Event-B models regarding behavioral traces in three steps. In the first step, we provided the trace semantics for trigger-response properties and real-time trigger-response properties. Then we provided sufficient conditions as well as fairness assumptions for the traces of Event-B machine to satisfy such properties. In the second step, we explored additional conditions to verify refinement in Event-B models in terms of infinite behavioral traces. Zorn's Lemma was used together with forward simulation to prove infinite state/event trace refinement in refinement steps. In the last step, we used trace semantics to explore additional proof obligations to refine timing properties in Event-B models. Based on the generic refinement semantics and theorems of Event-B models in terms of infinite traces, we used the bounded retransmission protocol case study in Chapter 5 to demonstrate the refinement conditions.

In Chapter 6, we provided four real-time specification patterns to represent time in Event-B models. We presented the specification patterns with Event-B syntax to model real-time properties in real-world cases. Our patterns enable the modelers to express real-time properties that can be verified with the Event-B theorem proving technique. Also, we presented refinement patterns to refine the abstract timing properties modeled with different specification patterns into concrete sub-timing properties. With the cardiac pacemaker case study in Chapter 7, we showed that the complicated timing cycles in pacemakers could be developed in a stepwise manner by combining different real-time specification patterns. The proof results of the model also showed that the usage of patterns could help to discharge the Event-B proof obligations automatically in the Rodin tool.

In Chapter 8, we presented the syntax of parameterized real-time trigger-response properties with Event-B formalization and proofs based on a trigger-response approach to modeling timing properties in Event-B. Rules and proofs were provided to replace parameterized timing properties with unparameterized timing properties. To distinguish timing properties from the perspective of different system design phases, we defined parameterized timing properties that place discrete-time constraints on individual tasks as end-to-end timing properties, which described high-level timing properties from the system requirement specification phase. These end-to-end timing properties cannot precisely describe the concurrent behavior of tasks. To model the behavior of these concurrent tasks, we defined scheduler-based timing properties with unparameterized timing properties as concrete timing properties for the system design phase, which places discrete-time constraints on the scheduler that schedules the concurrent tasks. To replace end-to-end timing properties with scheduler-based timing properties, we introduced a nondeterministic queue-based scheduling policy with some additional gluing invariants. Then in Chapter 9, we formalized a two-level hierarchical scheduling system that refines the nondeterministic queue-based scheduling policies to two compatible local scheduling policies to illustrate the pattern that replaces end-to-end timing properties with scheduler-based timing properties.

10.2 Comparison with Related Work

Several researchers have explored approaches to modeling real-time properties in Event-B models. We overview these approaches in section 2.4.4.6. In this section, we mainly describe the contributions we made to the work done regarding Event-B.

The work of Cansell et al. (2007) does not provide the refinement patterns for timing properties. Our work not only presents real-time specification patterns to model time in CPS but also provides refinement patterns to develop the complex real-time systems in a stepwise approach.

The work of Sarshogh and Butler (2011) classifies the timing constraints into delay, deadline and expiry and provides refinement patterns to refine the time constraints. However, their proposed trigger-response pattern lacks a proper treatment of the divergence of intermediate events. *Zeno* behavior could occur because of infeasible responses or conflicting timing constraints. Our work addresses these problems with additional proof obligations and fairness assumptions regarding intermediate events and response events. We also provide additional conditions for the refinement of timing properties relating to infinite behavioral traces. We show how to encode these conditions in Event-B models syntactically in the way that standard Event-B proof obligations are defined. In addition, we provide refinement rules for replacing parameterized timing properties with unparameterized timing properties.

The work of Sulskus et al. (2016) models the time constraints in Event-B models as a high-level abstraction in terms of the state machine. Their approach produces a finite-state space model that allows model checking for the timing properties in Event-B models. However, our approach models the infinite state-space with the unbounded *clock* variable that models the time. Proofs are provided to show that time will always progress with the fairness assumptions and additional conditions imposed on Event-B models.

10.3 Scope

Our approach, which extends Event-B models with real-time properties, mainly deals with discrete-time properties, while the real-world events do not always happen at integer-value times. Continuous-time can be modeled approximately by choosing the global clock granularity, which models the timed system with an approximate sense. There is research that introduces differential equations to model continuous behavior and discrete behavior in Event-B models. Banach et al. (2015) devised a Hybrid Event-B extension that accommodates continuous behaviors in between discrete transitions. Based on this extension, Butler et al. (2016) outlines an approach to modeling and reasoning about hybrid systems that uses continuous functions over real intervals to model the evolution of continuous values over time.

In addition, our theories regarding the trigger-response properties and real-time trigger-response properties rely on the weak fairness assumptions of specific events. We did not investigate the fairness refinement rules to prove liveness properties. However, TLA allows the notion of refinement rules for strong fairness and weak fairness (Méry and Poppleton, 2015). An integration of TLA and Event-B could be used to verify real-time properties and liveness properties.

10.4 Future Work

Our work introduces additional proof obligations required to model discrete timing properties, which are not supported natively by the Rodin platform. A plugin can be built to extend the Rodin platform to support discrete-time modeling with trigger-response properties as well as conditional convergent event modeling in Event-B. Based on the monotonicity of timing properties, the plugin could extend the Event-B models to the corresponding timed model with the four real-time specification patterns. The plugin could also support the sequential refinement rules and alternative refinement rules to refine abstract timing properties to concrete timing properties.

To explicitly represent timing properties in a CPS, there are three typical time constraints to look into: period, deadline, worst-case execution time. More work can be done to apply some scheduling policies such as Rate-Monotonic (RM) and priority inheritance protocol based on the queue based scheduling framework to analyze the real-time performance of CPS together with the mentioned time constraints in Event-B. Hoang et al. (2011) proposed to reuse simple models as patterns to construct larger models. Our work that refines real-time properties with scheduling framework can also be used as patterns to refine complicated real-time systems with concurrent tasks.

Bibliography

Martín Abadi and Stephan Merz. On TLA as a logic. In Manfred Broy, editor, *Proceedings of the NATO Advanced Study Institute on Deductive Program Design, Marktoberdorf, Germany*, pages 235–271, 1996.

Martín Abadi and Leslie Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82(2):253–284, May 1991. ISSN 0304-3975.

Martín Abadi and Leslie Lamport. An old-fashioned recipe for real time. *ACM Trans. Program. Lang. Syst.*, 16(5):1543–1571, September 1994. ISSN 0164-0925, 1558-4593.

Nouha Abid, Silvano Dal-Zilio, and Didier Le Botlan. Real-time specification patterns and tools. *CoRR*, abs/1301.7534, 2013.

Nouha Abid, Silvano Dal Zilio, and Didier Le Botlan. Real-time specification patterns and tools. In *International Workshop on Formal Methods for Industrial Critical Systems*, pages 1–15. Springer, 2012.

Jean-Raymond Abrial. *Modeling in Event-B*. Cambridge University Press, 2009. ISBN 9781139195881.

Jean-Raymond Abrial, Michael Butler, Stefan Hallerstede, Thai Son Hoang, Farhad Mehta, and Laurent Voisin. Rodin: An open toolset for modelling and reasoning in Event-B. *Int J Softw Tools Technol Transfer*, 12(6):447–466, April 2010. ISSN 1433-2779, 1433-2787.

J.R. Abrial, A. Hoare, and Pierre Chapron. *The B-Book*. Cambridge University Press, October 1996. ISBN 9780521496193, 9780521021753, 9780511624162.

Syed Hassan Ahmed, Gwanghyeon Kim, and Dongkyun Kim. Cyber physical system: Architecture, applications and research challenges. In *2013 IFIP Wireless Days (WD)*, pages 1–5. IEEE, IEEE, November 2013. ISBN 9781479905430.

Bowen Alpern and Fred B. Schneider. Defining liveness. *Information Processing Letters*, 21(4):181–185, October 1985. ISSN 0020-0190.

R. Alur and T.A. Henzinger. Real-time logics: Complexity and expressiveness. In *[1990] Proceedings. Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 390–401. IEEE, IEEE Comput. Soc. Press, 1990. ISBN 0818620730.

Rajeev Alur. *Techniques for automatic verification of real-time systems.* PhD thesis, stanford university, 1991.

Rajeev Alur. *Principles of cyber-physical systems.* MIT Press, 2015.

Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, April 1994. ISSN 0304-3975.

Rajeev Alur and Thomas A Henzinger. Logics and models of real time: A survey. In *Workshop/School/Symposium of the REX Project (Research and Education in Concurrent Systems)*, pages 74–106. Springer, 1991.

Rajeev Alur and Thomas A. Henzinger. A really temporal logic. *J. ACM*, 41(1):181–203, January 1994. ISSN 0004-5411, 1557-735X.

Rajeev Alur and Thomas A. Henzinger. Finitary fairness. *ACM Trans. Program. Lang. Syst.*, 20(6):1171–1194, November 1998. ISSN 0164-0925.

Ralph-JR Back. Refinement calculus, part II: Parallel and reactive programs. In *Workshop/School/Symposium of the REX Project (Research and Education in Concurrent Systems)*, pages 67–93. Springer, 1989.

Richard Banach, Michael Butler, Shengchao Qin, Nitika Verma, and Huibiao Zhu. Core hybrid Event-B i: Single hybrid Event-B machines. *Science of Computer Programming*, 105:92–123, July 2015. ISSN 0167-6423.

S Serge Barold, Roland X Stroobandt, and Alfons F Sinnaeve. *Cardiac pacemakers and resynchronization step by step: An illustrated guide.* John Wiley & Sons, 2010.

Clark Barrett and Cesare Tinelli. Cvc3. In *International Conference on Computer Aided Verification*, pages 298–302. Springer, 2007.

Gerd Behrmann, Alexandre David, and Kim G. Larsen. A tutorial on uppaal. In *Lecture Notes in Computer Science*, pages 200–236. Springer Berlin Heidelberg, 2004. ISBN 9783540230687, 9783540300809.

Gerd Behrmann, Alexandre David, Kim Guldstrand Larsen, Paul Pettersson, and Wang Yi. Developing UPPAAL over 15 years. *Softw: Pract. Exper.*, 41(2):133–142, January 2011. ISSN 0038-0644.

Wolfgang Bibel. *Automated Theorem Proving.* Vieweg+Teubner Verlag, 1987. ISBN 9783528185206, 9783322901026.

Michael Butler. Incremental design of distributed systems with Event-B. In Manfred Broy, Wassiou Sitou, and Tony Hoare, editors, *Engineering Methods and Tools for Software Safety and Security - Marktoberdorf Summer School 2008*, pages 131–160. IOS Press, 2009. Chapter: 4.

Michael Butler. Mastering system analysis and design through abstraction and refinement. In *Engineering Dependable Software Systems*. IOS Press, 2013a.

Michael Butler. Mastering system analysis and design through abstraction and refinement. In *Engineering Dependable Software Systems*. IOS Press, 2013b.

Michael Butler. Reasoned modelling with Event-B. In Jonathan P. Bowen, Zhiming Liu, and Zili Zhang, editors, *Engineering Trustworthy Software Systems*, volume 10215, pages 51–109. Springer International Publishing, April 2017. ISBN 9783319568409, 9783319568416. Lecture notes for Spring School on Engineering Trustworthy Software Systems 2016, Chongqing, China.

Michael Butler, Jean-Raymond Abrial, and Richard Banach. Modelling and refining hybrid systems in Event-B and Rodin. In Luigia Petre and Emil Sekerinski, editors, *From Action Systems to Distributed Systems*, pages 29–42. Chapman and Hall/CRC, April 2016. ISBN 9781498701587, 9781498701594.

Michael Butler and Jerome Falampin. An approach to modelling and refining timing properties in b. In *Refinement of Critical Systems (RCS)*, January 2002.

M.J. Butler. *A CSP Approach to Action Systems*. PhD thesis, Oxford University, 1992.

Dominique Cansell, Dominique Méry, and Joris Rehm. Time constraint patterns for Event-B development. In *International Conference of B Users*, pages 140–154. Springer, 2007.

Alok Choudary, Vijay Geholt, and B. Narahari. Software design for real-time systems on parallel computers: Formal specifications. 1996.

Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In *International Conference on Computer Aided Verification*, pages 154–169. Springer, 2000.

Edmund M. Clarke, Orna Grumberg, and David E. Long. Model checking and abstraction. *ACM Trans. Program. Lang. Syst.*, 16(5):1512–1542, September 1994. ISSN 0164-0925, 1558-4593.

Edmund M Clarke, William Klieber, Miloš Nováček, and Paolo Zuliani. Model checking and the state explosion problem. In *LASER Summer School on Software Engineering*, pages 1–30. Springer, 2011.

Edmund M. Clarke and Jeannette M. Wing. Formal methods: State of the art and future directions. *CSUR*, 28(4):626–643, December 1996. ISSN 0360-0300.

ClearSy. Atelier B, 2019. Available online at `http://www.atelierb.eu/` [Accessed: 9 Mar 2019].

Keith Conrad. Zorn's lemma and some applications. *Expository papers*, 2016.

J.E. Coolahan and N. Roussopoulos. Timing requirements for time-driven systems using augmented petri nets. *IIEEE Trans. Software Eng.*, SE-9(5):603–616, September 1983. ISSN 0098-5589.

Pedro R D'Argenio, J-P Katoen, Theo C Ruys, and Jan Tretmans. The bounded retransmission protocol must be on time! In *International Workshop on Tools and Algorithms for the Construction and Analysis of Systems*, pages 416–431. Springer, 1997.

Leonardo De Moura and Nikolaj Bjorner. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

Nachum Dershowitz, D.N. Jayasimha, and Seungjoon Park. Bounded fairness. In *Lecture Notes in Computer Science*, pages 304–317. Springer Berlin Heidelberg, 2003. ISBN 9783540210023, 9783540399100.

Keith J. Devlin. *Fundamentals of Contemporary Set Theory*. Springer US, 1979. ISBN 9780387904412, 9781468400847.

Henning Dierks, Sebastian Kupferschmid, and Kim G Larsen. Automatic abstraction refinement for timed automata. In *International Conference on Formal Modeling and Analysis of Timed Systems*, pages 114–129. Springer, 2007.

Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Patterns in property specifications for finite-state verification. In *Proceedings of the 21st international conference on Software engineering - ICSE '99*, pages 411–420. IEEE, ACM Press, 1999. ISBN 1581130740.

Event B.org. Event-B and the Rodin Platform, 2019a. Available online at `http://www.event-b.org` [Accessed: 9 Mar 2019].

Event B.org. Rodin Platform, 2019b. Available online at `http://wiki.event-b.org/index.php/Rodin_Platform` [Accessed: 9 Mar 2019].

Nissim Francez. *Fairness*. Springer US, 1986. ISBN 9781461293477, 9781461248866.

V. Gehlot. Performance specification and livelock detection/correction of a protocol using timed petri nets. In *IEEE International Conference on Communications, - Spanning the Universe.*, pages 1286–1290. IEEE, IEEE, 1988.

Mario Gleirscher, Simon Foster, and Jim Woodcock. New opportunities for integrated formal methods. *ACM Computing Surveys (CSUR)*, 52(6):1–36, 2019.

Stefan Hallerstede, Michael Leuschel, and Daniel Plagge. Validation of formal models by refinement animation. *Science of Computer Programming*, 78(3):272–292, March 2013. ISSN 0167-6423.

Thomas A Henzinger. Temporal specification and verification of real-time systems. Technical report, STANFORD UNIV CA DEPT OF COMPUTER SCIENCE, 1991.

Thomas A Henzinger, Zohar Manna, and Amir Pnueli. Timed transition systems. In *Workshop/School/Symposium of the REX Project (Research and Education in Concurrent Systems)*, pages 226–251. Springer, 1991.

Andrew Hinton, Marta Kwiatkowska, Gethin Norman, and David Parker. PRISM: A tool for automatic verification of probabilistic systems. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 441–444. Springer, 2006.

Thai Son Hoang. How to interpret failed proofs in Event-B. *Technical report*, 672, 2010.

Thai Son Hoang, Andreas Fürst, and Jean-Raymond Abrial. Event-B patterns and their tool support. *Softw Syst Model*, 12(2):229–244, January 2011. ISSN 1619-1366, 1619-1374.

J Holzmann Gerard. SPIN model checker: The primer and reference manual, 2003.

M. Jastram and P.M. Butler. *Rodin User's Handbook: Covers Rodin V.2.8*. 2.8covers Rodin. Createspace Independent Pub, 2014. ISBN 9781495438141.

Zhihao Jiang and Rahul Mangharam. Modeling cardiac pacemaker malfunctions with the virtual heart model. In *2011 Annual International Conference of the IEEE Engineering in Medicine and Biology Society*, pages 263–266. IEEE, IEEE, August 2011. ISBN 9781457715891, 9781424441211, 9781424441228.

Zhihao Jiang, Miroslav Pajic, Rajeev Alur, and Rahul Mangharam. Closed-loop verification of medical devices with model abstraction and refinement. *Int J Softw Tools Technol Transfer*, 16(2):191–213, September 2013. ISSN 1433-2779, 1433-2787.

Zhihao Jiang, Miroslav Pajic, and Rahul Mangharam. Model-based closed-loop testing of implantable pacemakers. In *2011 IEEE/ACM Second International Conference on Cyber-Physical Systems*, pages 131–140. IEEE, IEEE, April 2011. ISBN 9781612846408.

Zhihao Jiang, Miroslav Pajic, Salar Moarref, Rajeev Alur, and Rahul Mangharam. Modeling and verification of a dual chamber implantable pacemaker. In Cormac Flanagan and Barbara König, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 188–203, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg. ISBN 978-3-642-28756-5.

He Jifeng. Process simulation and refinement. *Formal Aspects of Computing*, 1(1): 229–241, March 1989a. ISSN 0934-5043, 1433-299X.

He Jifeng. Various simulations and refinements. In *Workshop/School/Symposium of the REX Project (Research and Education in Concurrent Systems)*, pages 340–360. Springer, 1989b.

Taylor Johnson and Sayan Mitra. Handling failures in cyber-physical systems: Potential directions. In *The Real-Time Systems Symposium*. Citeseer, 2009.

Mark B. Josephs. A state-based approach to communicating processes. *Distrib Comput*, 3(1):9–18, March 1988. ISSN 0178-2770, 1432-0452.

Robert M. Keller. Formal verification of parallel programs. *Commun. ACM*, 19(7): 371–384, July 1976. ISSN 0001-0782.

Jai-Hoon Kim, SH Park, and Geoffery Fox. Real-time scheduling in cyber-physical systems. In *International Conference, CCA*, page 69, 2012.

John C. Knight. Safety critical systems. In *Proceedings of the 24th international conference on Software engineering - ICSE '02*, pages 547–550. ACM, ACM Press, 2002. ISBN 158113472X.

Sascha Konrad and Betty H.C. Cheng. Real-time specification patterns. In *Proceedings of the 27th international conference on Software engineering - ICSE '05*, pages 372–381. ACM, ACM Press, 2005. ISBN 1595939632.

Hermann Kopetz. Design principles for distributed embedded application, 1997.

Hermann Kopetz. *Real-Time Systems*. Springer US, 2011. ISBN 9781441982360, 9781441982377.

L. Lamport. Proving the correctness of multiprocess programs. *IIEEE Trans. Software Eng.*, SE-3(2):125–143, March 1977. ISSN 0098-5589.

Leslie Lamport. *Specifying systems: The TLA+ language and tools for hardware and software engineers*. Addison-Wesley Longman Publishing Co., Inc., 2002.

Kim G. Larsen, Paul Pettersson, and Wang Yi. Uppaal in a nutshell. *STTT*, 1(1-2): 134–152, December 1997. ISSN 1433-2779.

G. Le Lann. An analysis of the ariane 5 flight 501 failure-a system engineering perspective. In *Proceedings International Conference and Workshop on Engineering of Computer-Based Systems*, ECBS'97, pages 339–346, Washington, DC, USA, 1997. IEEE Computer. Soc. Press. ISBN 0818678895.

Barbara Liskov, John Guttag, et al. *Abstraction and specification in program development*, volume 180. MIT press Cambridge, 1986.

C.L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20(1):46–61, January 1973. ISSN 0004-5411, 1557-735X.

N. Lynch and F. Vaandrager. Forward and backward simulations. *Information and Computation*, 121(2):214–233, September 1995. ISSN 0890-5401.

Dominique Méry, Neeraj Kumar Singh, et al. Functional behavior of a cardiac pacing system. *International Journal of Discrete Event Control Systems*, 1(2):129–149, 2011.

Robin Milner. *An algebraic definition of simulation between programs*. Citeseer, 1971.

L.E. Moser, Y.S. Ramakrishna, G. Kutty, P.M. Melliar-Smith, and L.K. Dillon. A graphical environment for the design of concurrent real-time systems. *ACM Trans. Softw. Eng. Methodol.*, 6(1):31–79, January 1997. ISSN 1049-331X, 1557-7392.

Dominique Méry and Michael Poppleton. Towards an integrated formal method for verification of liveness properties in distributed systems: With application to population protocols. *Softw Syst Model*, 16(4):1083–1115, December 2015. ISSN 1619-1366, 1619-1374.

Gethin Norman, David Parker, and Jeremy Sproston. Model checking for probabilistic timed automata. *Form Methods Syst Des*, 43(2):164–190, October 2012. ISSN 0925-9856, 1572-8102.

Nsf. Cyber-physical systems (CPS), 2019.

Jonathan S Ostroff. *Temporal logic for real-time systems*, volume 40. Research Studies Press Advanced Software Development Series, 1989.

Jonathan S. Ostroff. Formal methods for the specification and design of real-time safety critical systems. *Journal of Systems and Software*, 18(1):33–60, April 1992. ISSN 0164-1212.

Jonathan S. Ostroff. Composition and refinement of discrete real-time systems. *ACM Trans. Softw. Eng. Methodol.*, 8(1):1–48, January 1999. ISSN 1049-331X, 1557-7392.

Lawrence C Paulson. *Isabelle*, volume 828. Springer-Verlag, 1994. ISBN 3540582444.

Amir Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*, pages 46–57. IEEE, IEEE, September 1977.

Amir Pnueli and Eyal Harel. Applications of temporal logic to the specification of real-time systems. In *Systems, Proceedings of a Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 84–98, London, UK, UK, 1988. Springer-Verlag. ISBN 3-540-50302-1.

Michael Poppleton and Abdolbaghi Rezazadeh. Modelling the pacemaker in event-b: Towards methodology for reuse. September 2012.

Ken Robinson. System modelling & design using Event-B. *The University of New South Wales. Recuperado en agosto de*, 2012.

Kristin Y. Rozier. Linear temporal logic symbolic model checking. *Computer Science Review*, 5(2):163–203, May 2011. ISSN 1574-0137.

Mohammad Reza Sarshogh. *Extending Event-B with discrete timing properties*. PhD thesis, University of Southampton, May 2013.

Mohammad Reza Sarshogh and Michael Butler. Specification and refinement of discrete timing properties in Event-B. In *AVoCS 2011*, September 2011. Event Dates: September 2011.

Steve Schneider, Helen Treharne, and Heike Wehrheim. The behavioural semantics of Event-B refinement. *Form Asp Comp*, 26(2):251–280, October 2012. ISSN 0934-5043, 1433-299X.

Boston Scientific. Pacemaker system specification. *Boston Scientific*, 2007.

Emil Sekerinski and Tian Zhang. Finitary fairness in action systems. In *International Colloquium on Theoretical Aspects of Computing*, pages 319–336. Springer, 2013.

L. Sha, R. Rajkumar, and J.P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Trans. Comput.*, 39(9):1175–1185, 1990. ISSN 0018-9340.

Jianhua Shi, Jiafu Wan, Hehua Yan, and Hui Suo. A survey of cyber-physical systems. In *2011 International Conference on Wireless Communications and Signal Processing (WCSP)*, pages 1–6. IEEE, IEEE, November 2011. ISBN 9781457710100, 9781457710094, 9781457710070, 9781457710087.

John A. Stankovic, Marco Spuri, Krithi Ramamritham, and Giorgio C. Buttazzo. *Deadline Scheduling for Real-Time Systems*. Springer US, 1998. ISBN 9781461375302, 9781461555353.

Eugene William Stark. Foundations of a theory of specification for distributed systems. 1984.

Gintautas Sulskus. *An investigation into Event-B methodologies and timing constraint modelling*. PhD thesis, University of Southampton, September 2017.

Gintautas Sulskus, Michael Poppleton, and Abdolbaghi Rezazadeh. Modelling complex timing requirements with refinement. In *2016 IEEE 17th International Conference on Information Reuse and Integration (IRI)*, volume 9392, pages 292–307. IEEE, July 2016. ISBN 9781509032075.

Gregory Tassey. The economic impacts of inadequate infrastructure for software testing. *National Institute of Standards and Technology, RTI Project*, 7007(011):429–489, 2002.

Dolores R. Wallace and D. Richard Kuhn. Failure modes in medical device software: An analysis of 15 years of recall data. *Int. J. Rel. Qual. Saf. Eng.*, 08(04):351–371, December 2001. ISSN 0218-5393, 1793-6446.

Jim Woodcock, Peter Gorm Larsen, Juan Bicarregui, and John Fitzgerald. Formal methods. *ACM Comput. Surv.*, 41(4):1–36, October 2009a. ISSN 0360-0300, 1557-7341.

Jim Woodcock, Peter Gorm Larsen, Juan Bicarregui, and John Fitzgerald. Formal methods. *ACM Comput. Surv.*, 41(4):1–36, October 2009b. ISSN 0360-0300, 1557-7341.

Sergio Yovine. KRONOS: A verification tool for real-time systems. *STTT*, 1(1-2):123–133, December 1997. ISSN 1433-2779.

Yuan Yu, Panagiotis Manolios, and Leslie Lamport. Model checking TLA+ specifications. In *Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, pages 54–66. Springer, 1999.

Hehua Zhang, Ming Gu, and Xiaoyu Song. Specifying time-sensitive systems with TLA+. In *2010 IEEE 34th Annual Computer Software and Applications Conference*, pages 425–430. IEEE, IEEE, July 2010. ISBN 9781424475124.

Chenyang Zhu, Michael Butler, and Corina Cirstea. Semantics of real-time trigger-response properties in Event-B. In *2018 International Symposium on Theoretical Aspects of Software Engineering (TASE)*, pages 150–155. IEEE, August 2018a. ISBN 9781538673058.

Chenyang Zhu, Michael Butler, and Corina Cirstea. Formalizing hierarchical scheduling for refinement of real-time systems, September 2019a.

Chenyang Zhu, Michael Butler, and Corina Cirstea. Formalizing hierarchical scheduling for refinement of real-time systems. *Science of Computer Programming*, 189:102390, April 2020a. ISSN 0167-6423.

Chenyang Zhu, Michael Butler, and Corina Cirstea. Real-time trigger-response properties for Event-B applied to the pacemaker. In *The 14th International Symposium on Theoretical Aspects of Software Engineering*. IEEE, March 2020b.

Chenyang Zhu, Michael Butler, and Corina Cirstea. Trace semantics and refinement patterns for real-time properties in event-b models. *Science of Computer Programming*, page 102513, 2020c. ISSN 0167-6423.

Chenyang Zhu, Michael J. Butler, and Corina Cîrstea. Refinement of timing constraints for concurrent tasks with scheduling. In Michael J. Butler, Alexander Raschke, Thai Son Hoang, and Klaus Reichl, editors, *Abstract State Machines, Alloy, B, TLA, VDM, and Z - 6th International Conference, ABZ 2018, Southampton, UK, June*

5-8, 2018, Proceedings, volume 10817 of *Lecture Notes in Computer Science*, pages 219–233. Springer, 2018b.

Chenyang Zhu, Michael J. Butler, and Corina Cirstea. Towards refinement semantics of real-time trigger-response properties in Event-B. In Dominique Méry and Shengchao Qin, editors, *2019 International Symposium on Theoretical Aspects of Software Engineering, TASE 2019, Guilin, China, July 29-31, 2019*, pages 1–8. IEEE, 2019b.