UNIVERSITY OF SOUTHAMPTON

FACULTY OF PHYSICAL AND APPLIED SCIENCES

Electronics and Computer Science

Extending Event-B with Discrete Timing Properties

by

Mohammad Reza Sarshogh

Thesis for the degree of Doctor of Philosophy

May 2013

UNIVERSITY OF SOUTHAMPTON

ABSTRACT

FACULTY OF PHYSICAL AND APPLIED SCIENCES
Electronics and Computer Science

Doctor of Philosophy

EXTENDING EVENT-B WITH DISCRETE TIMING PROPERTIES

by Mohammad Reza Sarshogh

Event-B is a formal language for systems modelling, based on set theory and predicate logic. It has the advantage of mechanized proof, and it is possible to model a system in several levels of abstraction by using refinement. Discrete timing properties are important in many critical systems. However, modelling of timing properties is not directly supported in Event-B. In this work, we identify three main categories of discrete timing properties for trigger-response patterns, *deadline*, *delay* and *expiry*.

We introduce language constructs for each of these timing properties that augment the Event-B language. We describe how these constructs have been given a semantics in terms of the standard Event-B constructs. To ease the process of using timing properties in a refinement-based development, we introduce patterns for refining the timing constructs that allow timing properties on abstract models to be replaced by timing properties on refined models. The language constructs and refinement patterns are illustrated through some generic examples.

We have developed a tool to support our approach. Our tool is a plug-in to the Rodin tool-set for Event-B and automates the translation of timing properties to Event-B as well as the generation of gluing invariants, required to verify the consistency of timing properties refinement.

In the end, we demonstrate the practicality of our approach, by going through the modelling and verifying process of two real-time case studies. The main focus will be the usefulness of the timing refinement patterns in a step-wise modelling and verification process of a real-time system.

Contents

List of Figures

List of Tables

Acknowledgements

It is with immense gratitude that I acknowledge the support and help of my supervisor, Professor Michael Butler. Apart from supporting me throughout my PhD with his kindness and knowledge, our meetings were always enlightening the problem domain for me, and proving me with the sufficient guidance to investigate the best possible solution.

This research would not have been possible without the financial support of DEPLOY project. Besides, in my daily work I have been blessed with a friendly and cheerful group of fellow colleges in ESS research group.

In the end I cannot find words to express my gratitude to my parents and brother for their continues support and encouragement.

To my mother and father for their continuous support.

Nomenclature

GCL	The abbreviation of Guarded Command-language
CSP	The abbreviation of Communicating Sequential Processes
IDE	The abbreviation of Integrated Development Environment
MUI	The abbreviation of Proof Modelling User Interface
PUI	The abbreviation of Proof Proving User Interface
JSD	The abbreviation of Jackson System Development
LTL	The abbreviation of Linear-time Logic
CTL	The abbreviation of Computation Tree Logic
$TCTL$	The abbreviation of Timed Computation Tree Logic
HTA	The abbreviation of Hierarchical Timed Automata
VDM	The abbreviation of Vienna Development Method
PO	The abbreviation of Proof Obligation

Chapter 1

Introduction

Computers are rapidly becoming an important part of everyday life. Nowadays many infrastructures of our civilization partially or completely depend on information systems and we let computers manage and control most of our safety critical systems. A failure of these systems can end up tragically for their users and owners. As a result, reliability and accuracy of these systems are very important.

A *software failure* is a condition that causes a system to fail in performing its required functionalities [79]. Most software failures are caused by humans, during the design and implementation phases, and a few are caused by compilers. Mistakes are inevitable in humans' activities. It is possible to reduce human errors by changing the work environment and increasing the concentration of the user, but it is impossible to eliminate it.

Testing is defined as activities which aim at evaluating an attribute or capability of a program or a system to determine whether it meets its requirements [36]. Currently, in IT industry, the common approach is the problem testing which does not guarantee a faultless system, since in a real-size system, it is usually not possible to test all the possible scenarios. In a software system, a solution validation, is only concerned about validating some software properties (e.g., null pointers, overflow, etc.) of a constructed software [13]. Many experts in this area such as Abrial [13] and Jackson [78] believe in the problem verification as a way of developing faultless systems. Based on the problem verification approach, the goal is to verify the software as a part of a system. So, the environment in which the software is preforming is as important as the software itself, in the verification process.

In the problem verification, modelling of a system becomes an important development phase in order to validate the overall purpose of that system [14]. Event-B [13, 6] is a step-wise formal modelling language which uses set theory as the modelling notation

for system level modelling and analysis. The idea is to develop a method which combines modelling and validation, uses refinement to represent systems in different level of abstraction, and verifies their consistency by mathematical proof.

Most of safety-critical systems are *real-time*. In real-time systems it is not enough to just have a correct reaction to a user's requests or the environment changes, but the correct reaction, should happen in a specific period of time. Hence, the specifications of real-time systems, include many time related requirements. Time plays an important role in *distributed embedded systems*. Embedded systems are computing systems, intimately coupled to their target environments, which they monitor and control [111].

A large portion of a real-time system development costs is devoted to ensure the product is *fit-for-purpose* [81]. Time is one of the things, which makes this process more complex for real-time systems than others. The time-based scheduling of real-time systems, adds extra details to the ordering of events' occurrences. So extra properties are required to be verified about a real-time system's behaviour.

1.1 The Contribution

Event-B lacks explicit support for expressing and verifying timing properties. In a time-critical system, timing properties specify timing boundaries on the system reactions and responses.

Modelling time-critical systems, using Event-B has been investigated in several studies. Our contribution is the categorization of the discrete timing properties in three groups: deadline, delay and expiry, augmenting Event-B with some language constructs for them, introducing some patterns for refining timing properties and proving the consistency of their refinements. Plus, investigating the decomposition process of timed Event-B models. Also, a plug-in has been developed to support the approach (adding timing properties and refining them) in the Rodin tool-set.

Defining a semantics based on the standard Event-B constructs for timing properties, helped us to benefit from the existing features of Event-B such as refinement and decomposition, with no change or adaptation required.

Event-B refinement allows atomic events at the abstract level to be broken down into sub-steps at the concrete level. The goal of our refinement patterns is to provide an easy way to model the timing properties on the abstract atomic events, and then correctly refine them, with more elaborate timing properties on the concrete events.

In any system analysis, there always exists a level of granularity which will not be broken to a finer one. In a time-critical system, there are assumptions about the maximum

duration required to establish each step at the concrete level such as individual assignments or signal transmissions. Based on those assumptions it is possible to analyse the required time for a composite process to respond to a request or react to a change in that system. In our approach there is a top-bottom analysis. So, we start from the abstract behaviours of a system and model their properties and timing assumptions, then by each refinement, we introduce the sub-steps of the abstract behaviours and replace their timing assumptions with the timing assumptions of the sub-steps. As a result, by verifying the consistency of the refinement, we prove the consistency of the timing assumptions in different levels of abstraction. Hence, when the modeller verifies the consistency of the last refinement, he/she has verified the satisfaction of the abstract timing properties based on the timing assumptions of the most concrete behaviours.

In order to evaluate our approaches, two real-time case studies have been selected to be modelled in Event-B, and the result is presented in this report.

1.2 Thesis Roadmap

In the following chapter, the concepts required for following the discussion of modelling real-time systems in Event-B, will be explained.

In Chapter 3, some of the existing works on modelling real-time systems and reasoning about them, will be introduced briefly.

In Chapter 4, the discrete timing properties we want to extend Event-B with will be introduced and their semantics will be presented and proved. Besides, their refinement will be explained in forms of some refinement patterns.

In Chapter 5 decomposition of timed Event-B models will be explained.

In Chapter 6, the effect of adding timing properties on events' enableness will be discussed.

In Chapter 7 the automatic gear controller case study will be explained, before introducing parametrized timing properties in Chapter 8. In Chapter 8 the syntax and semantics of parametrized timing properties will be discussed and some of their refinement patterns will be explained.

In Chapter 9 a case study will be discussed which aims to evaluate the practicality of parametrized timing properties.

In Chapter 10, the plug-in developed for the Rodin tool-set, to support discrete timing properties extension will be discussed. Finally, in Chapter 11 our conclusions will be presented.

Chapter 2

Background on Formal Reasoning

Refinement, decomposition and the consistency verification of an Event-B model, play an important role in this work. Event-B inherited these features from its predecessor formal languages. As a result, in this chapter we will go through some of these languages which their modelling and verification approaches affected the existing modelling and verification features of Event-B. Beside, Event-B and Refinement Diagrams will be discussed to provide the required background to follow the discussion of extending Event-B by timing properties.

Section 2.1 looks at Dijkstra's Guarded Command-language. Section 2.2 looks at the refinement calculus which has been developed for Dijkstra's Guarded Command-language to transform it to a runnable code. Section 2.3 talks about temporal logic which plays an important role in specifying reactive systems' properties. Section 2.4 discusses transition systems, which are the state-based modelling approaches. Section 2.5 looks at action systems, which have had a great influence on Event-B method. Section 2.6 talks about communicating sequential processes method, which is an event-based modelling approach and its parallel composition has inspired share-event decomposition in Event-B. Section 2.7 looks at some approaches to bridge between event and state based modelling approaches, which is one of the main Event-B targets. Section 2.8 looks at Event-B method and some of its key features. Section 2.9 talks about existing tool support for Event-B method. Finally, Section 2.10 talks about refinement diagram notation, which is helpful in constructing atomicity decomposition refinements in Event-B.

2.1 Dijkstra's Guarded Command-language

As mentioned before, action systems have some major influences on modelling approaches of Event-B. But since action systems are partially based on Dijkstra's guarded command-language (GCL) [55], before we talk about them, it is useful to briefly introduce GCL and its associated weakest-precondition calculus.

Programs in GCL have three main characteristics, they act on variables, they are sequential, and they are intended to terminate. The behaviour of a program is specified by describing a condition of the initial values of the program variables, as precondition, and a condition of their final values, as postcondition.

Dijkstra has introduced weakest precondition semantics to deal with the termination of programs' models. Based on weakest precondition semantics for a statement S and a postcondition $post$, $wp(S, post)$ represents all those initial states from which S is guaranteed to terminate in a state which satisfies the $post$ condition [43]. As a result, statement S satisfies a specification $(pre, post)$ if:

$$pre \Rightarrow wp(S, post).$$

Based on the notion of total correctness in GCL, a statement S is totally correct with respect to precondition p, and postcondition q, if it guarantees to terminate in a state that satisfies q, whenever the initial state satisfies p. Besides, Dijkstra presented a calculus for verifying the satisfaction of a programme's specifications in GCL which can be found in [55].

The syntax of GCL can be expressed as follow based on [59]:

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

Where the braces {..} should be read as *followed by zero or more instances of the enclosed*, *"other statements"* can be assignment statements and procedure calls, and the semicolons specify the order of statements' execution in a guarded list. When a guarded list is selected for execution, its statements will be executed successively from left to right [59]. If a guarded command set, consists of more than one statement, statements will be executed in a non-deterministic order. This arbitrarily order of execution has been presented by the separator *[]* in the syntax.

Action systems have inherited the semicolons and loop constructs from GCL, but guards have replaced the pre-conditions which will be explained in more details in Section 2.5.

As explained, emphasis in GCL is the termination of programs' models. In timed Event-B, some of the timing properties, such as deadline, act as the local termination conditions. This feature will be discussed in Chapter 4.

2.2 Refinement Calculus

Since the refinement calculus has affected the refinement approach in action systems, a short introduction to it will be presented in this section based on the Back's book [31]. Refinement calculus is a logical framework for reasoning about program correctness introduced by Back and Wright in [29, 24, 92]. It focuses on two main questions:whether a program is correct based on a given specification, and how it can be improved, or refined, while preserving its correctness.

Refinement calculus has been originated from stepwise refinement method for program construction by Dijkstra [53] and Wirth [112], the transformational approach to programming [66, 39], and the early works of Hoare on program correctness and data refinement [71, 72], in which data refinement transforms a program of one data type to another [91]. The purpose of the refinement calculus is to provide a solid logical framework for all of these methods, based on Dijkstra's weakest pre-condition approach to total correctness of programs [54].

Consider the components of a system, where each of them can change the system state in different ways. What regulates the behaviour of these components and their cooperation is called a *contract*. So a contract can specify the order of actions that a component has to carry out. A program usually consists of a collection of interacting components. When we program a specific component, we assume that other components are controlled by other agents. A specification of a program component is a contract in which some constraints on that component's behaviour have been declared, without constraining the subcontractor or the implementer on how the actual behaviour of the component may be realized.

In order to define correctness, assume a contract statement S, a pre-condition p, and a post-condition q, then S is correct with respect to p and q, denoted by $p \ \{|\ S\ |\}\ q$ if for every σ that satisfies p, then $\sigma \ \{|\ S\ |\}\ q$. Since program statements are special kinds of contract statements, this definition of correctness can also be applied for program statements.

Based on the definition of program statement correctness, $p \ \{|\ S\ |\}\ q$ expresses that for any initial state in p, the agent can choose any execution of S that either satisfies q, or leads to the violation of some of its assumptions.

The main application of refinement calculus is to prove the correctness preservation of the stepwise refinement of a program that satisfies a given specification. In stepwise refinement, we start from a high-level specification of a program requirements. This specification is then replaced by program statements which implement what has been described by it, but contains some sub-specifications(not implemented) parts. This process is then repeated for those sub-specification, and it will continue until all the

specifications have been implemented by program statements, and an executable program has been produced. Refining a program is done by applying transformation, which changes the program in a way that preserve its correctness.

The refinement relation between contract statements is defined as follows:

$$S \sqsubseteq S' \ \wedge \ p \ \{|\ S\ |\}\ q \quad \Rightarrow \quad p \ \{|\ S'\ |\}\ q \tag{2.1}$$

and this is the case for any choice of p and q. As mentioned before, in stepwise refinement, we start with an initial statement S_0, which satisfies some correctness criteria. Then it will be evolved by a sequence of successive refinements

$$S_0 \sqsubseteq S_1 \sqsubseteq \cdots \sqsubseteq S_n \ . \tag{2.2}$$

Based on transitivity, we know $S_0 \sqsubseteq S_n$, and since the refinement preserves the correctness, it is guaranteed that S_n will satisfy the correctness criteria of S_0.

Refinement calculus is based on high-order logic and lattice theory, which provide the means to prove the correctness of programs and calculate a program refinements in a precise mathematically manner.

As it will be discussed in Section 2.8, we have correctness in Event-B which has been defined based on discharging proof obligations, and the correctness preservation by the concrete model has to be proved.

2.3 Temporal Logic

Originally, temporal logic was developed to be used in philosophy, and it has been proposed to be used in computer science by Burstall [38] and Pnueli [99]. By using temporal logic, it is possible to model dynamic behaviours in a simple fashion [93]. The main goal of using temporal logic in computer science is to appropriately formalizing the semantics of reactive systems [99].

Reactive systems [1] are computer systems that continually react to their environments at the speed in which their environments change. Some of their main features are, existence of concurrency, having strict timing requirements, and the importance of their reliability [2]. These systems have an ongoing interaction with their environments and their role is to maintain this interaction and to perform a desired computational role.

Because of this ongoing interaction, a language is required to describe the desirable behaviour of a reactive system, without referencing to its details of implementation [90]. Temporal logic defines predicates over infinite sequence of states, so it can be satisfied by some sequences and can be violated by some others. For example, consider a system with

two processes P_1 and P_2, where there is a mutually exclusion between them on a shared resource. Three examples of temporal properties for this system are as follows [90]:

- For all state of the system, it is never the case that P_1 and P_2 use the shared resource in a same time,
- Whenever P_1 wishes to use the shared resource, it will eventually do so,
- If there is a sequence where in position $j > 0$, P_2 is waiting to have access to shared resource, there is a position $s > j$ in that sequence where P_2 is using the shared variable (Another way of expressing the previous property for P_2).

As shown in the above examples, the expressed properties refer to all the possible execution sequences or some of them. So it is possible to specify the access policy for the shared resource between P_1 and P_2 independent of their detailed behaviours.

2.4 Transition Systems

Transition systems [100, 82] are typically used to model state-based reactive systems [43]. A transition system T may be defined in terms of a tuple (S, I, R) where S is a set of possible states, I is the subset of S and contains the initial states, and R is a set of transition relations. T starts in one of the specified initial states in I such as s_0. Then a transition relation will be selected from R such as r where $s_0 \in dom(r)$ and causes T to go to another state s_1, where $(s_0, s_1) \in r$. This process of selecting and occurrence of transition relations continues until T reaches a terminating state. Terminating states do not exist in any of the transition relations' domains.

Consider a state-trace st of a transition system $T = (S, I, R)$ as follows:

$$st = s_0 \rightarrow s_1 \rightarrow \cdots$$

Where $s_0 \in I$, and for each transition, there is a $r_i \in R$ $(i \geq 0)$ such that $(s_i, s_i + 1) \in r_i$. A state-trace such as st represents a possible behaviour of T. A transition system can be specified by expressing its properties. A property is a set of state-traces. A transition system T satisfies a property P, if all the state-traces of T exist in P. Properties can be categorized in two groups, liveness and safety properties. Safety properties specify that something bad will never happen, and liveness properties specify that something desirable will eventually happen. These properties can be specified using *temporal logic* instead of state-traces, as explained in Section 2.3.

2.4.1 Internal State

One of the challenges to model a reactive system is its complexity. A possible approach to deal with it, is abstraction. By hiding some of the details and specifying the system based on its high-level properties. This can be done in transition systems by hiding some parts of the state by using Abadi & Lamport's internal and external state notion [11]. Based on this approach each element of the state space is a pair of the form (e, i) where e is the external component and i is the internal component. As a result a state-trace will have the following form:

$$(e_0, i_0) \rightarrow (e_1, i_1) \rightarrow \cdots$$

In this way, a system specification, should describe the externally visible components of that system. However it is convenient to have the description of its unobservable internal components' behaviour. By having internal components, transition may be allowed in which just the internal state will be changed. Since for a transition system T, only the external behaviour is of interest, $tr_e(T)$ is assumed to strips away the internal components of the state-trace. As a result a state-trace in $tr_e(T)$ will look as follows:

$$e_0 \ \rightarrow e_1 \ \rightarrow \cdots \ . \tag{2.3}$$

By hiding internal components in a state-trace, some stuttering will appear, because of transitions between internal states. But finite stuttering in external state is not considered significant. Besides two external state-trace are equivalent if they are distinguishable only by finite number of stuttering [43].

2.4.2 Refinement

Property P' refines property P if $P' \subseteq P$. However, if properties are describe in temporal logic then the refinement can be proved by temporal proof rules such as those introduced by Manna and Pnueli in [89].

Proving the satisfaction of a property by a transition system is a form of refinement too. By separating properties into safety and liveness, it will be possible to prove safety properties by using invariance arguments, and liveness properties by well-foundedness arguments [17].

A transition system T' refines a transition system T if $tr_e(T') \subseteq tr_e(T)$. As a result, the external states of T' and T will be the same but their internal ones can be different by assuming the finite amount of stuttering.

Refinement mappings of transition systems are defined on more general components than those already explained. Other than (S, I, R) which have been presented before,

an extra component L is required too, which is a liveness property. So, the full state-traces of a transition system $T = (S, I, R, L)$ are defined as $tr(S, I, R) \cap L$ [43]. Function f from S' to S which preserves the external components, is a refinement mapping from T' to T if it satisfies the following conditions:

1. $f(I') \subseteq I$
2. $(s_0, s_1) \in R' \Rightarrow (f(s_0), f(s_1)) \in R \vee f(s_0) = f(s_1)$ (A transition in T' either corresponds to a transition in T, or it just causes a stutter)
3. $f(tr(T')) \subseteq L$.

Conditions 1 and 2 ensure satisfaction of the safety properties of T by T', which can be proved by reasoning about states and individual transitions. Condition 3 ensures that the liveness properties of T are satisfied by T'. Proving condition 3 is not as easy as conditions 1 and 2 because it involves state-traces.

2.5 Action Systems

Action systems are transition systems in which the state space may be represented by more than one variable, and the initialisation and transitions are represented by statements in Dijkstra's guarded command-language [43]. Since this thesis is about modelling approaches for Event-B, we will just talk about Back's action system [25] which Event-B has been influenced by.

In Back's action systems, an action (transition) is a guarded-command as follows:

$$g \rightarrow com,$$

where g is a condition on state variables and com is a program statement. An action is considered enabled if the state variables satisfy its guard. An action system starts by initialization and continues by selecting an enabled action and executing it. Actions are atomic and if several actions are enabled, one of them will be selected non-deterministically. Termination of an action system happens when it gets to a state where no action is enabled. Hence, based on Back's formalism, an action system is explicitly specified with an initialisation and a set of guarded-commands (actions).

An action system is a statement of the following form [32]

$$\mathcal{A} \quad :: \quad |[var\ x \cdot\ p;\ do\ A\ od]| : z \tag{2.4}$$

Where x and z are the tuples of local and global variables, p is their initialisation condition, and A is an action (can be a compound one). Since the action is inside a

loop, it is executed repeatedly. A is an atomic statement, so there will be no interruption in each of its iterations.

As explained the main difference of action systems and GCL, is that the preconditions have not been adopted by action systems, but we still have semicolons to specify the order of statements' execution. As it will explained in Section 2.8, there will be no semicolon in Event-B, and the order can be modelled by state variables. In this way, it will be possible to have invariants on the order of events' occurrences. By this introduction to the action systems, their refinement will be discussed in the following.

2.5.1 Action Systems Refinement

In order to show that one action system refines another in Back's formalism, the technique of data refinement for sequential programs is used [22, 24]. In data refinement, the aim is to prove that an abstract program $P(A)$ based on a data type A, is implemented correctly by a concrete program $P(C)$, where operations A_j from A are replaced by operations C_j, which are based on a more concrete data type C [52].

Similar to what has been explained in Section 2.4.1, the state variables of action systems are categorized into internal and external sets. Then if an action system T' refines another action system T, the internal state variables of T are regarded as the abstract variables, and the internal variables of T' as the concrete variables.

Based on Back's formalism, if Rep is a relation between the actions in $T' = (I', A')$ and $T = (I, A)$, then T' refines T under Rep, if [43]:

1. I is data refined by I' under Rep,
2. A is data refined by A' under Rep,
3. $Rep \ \wedge \ gd(A) \ \Rightarrow \ gd(A')$.

Where I and I' are the initial states, and A and A' represents the sets of actions in T and T'. Conditions 1 and 2 ensure that the concrete system preserve the safety properties of the abstract one. Data refinement is the transformation of a data type to another. Condition 3 ensures that T' terminates only if T terminates, which will guarantee the preservation of the liveness properties of T by T'.

Based on Back's refinement rules, a one to one correspondence between the actions of a concrete system and its abstract system's actions is required, in order to guarantee the preservation of the reactive behaviour of the abstract system by the concrete system's reactive behaviour. So it prevents any stuttering. But, this relation is more restrictive than what is usually needed [24]. So Back introduced a more general refinement rule which allows for stuttering actions in the concrete system. As a result, a concrete action

system T' is allowed to have some auxiliary actions H', which cause stuttering steps, as well as the main actions A'. In this way, $T' = (I', A', H')$ is a refinement of $T = (I, A)$ if besides holding conditions 1 and 2 as before, it also holds the following conditions [43]:

4. $Rep \wedge gd(A) \Rightarrow gd(A') \vee gd(H')$,
5. *skip* is data refined by H' under Rep,
6. Rep implies termination of *do* H' *od*.

Condition 4 ensure the termination of T' if and only if T terminates. Condition 5 ensures that actions in H' just cause stuttering transitions, while condition 6 ensures that only finite amount of stuttering can be caused by T'.

Based on Condition 3 the guards in the concrete system are stronger than the abstract ones, which can cause deadlock, because if an abstract guard holds, does not imply an enabled statement in the concrete machine. This issue has been resolved by replacing Condition 3 with Condition 4, where an enabled statement in the abstract system always imply one or more enabled statements in the concrete system.

Auxiliary actions have inspired *skip* events in Event-B which will be discussed in details in Section 2.8. As mentioned, auxiliary actions cause stuttering steps in the concrete system. Similarly, *skip* events refines skip in their abstract Event-B machine, since the transitions they present, are hidden in the abstraction.

2.5.1.1 Stepwise Refinement of Action Systems

In action systems, the behaviours of the parallel and distributed systems are modelled in terms of atomic actions. Atomicity of actions means if an action is executing, it is without interference of other actions in that system. Because of this characteristic, a parallel execution and a nondeterministic sequential execution of an action system have the same result[28]. As a result, it is possible to use the refinement calculus in order to model a parallel action system.

Based on this approach several actions can occur in parallel, if there is no common variables between them. There are two possible approaches to execute an action system in parallel, concurrent and distributed. In the case of a concurrent action system, actions are partitioned between processes. As a result, the communication and synchronization between actions in different processes is based on shared variable model. Any variable that is referenced by the actions of more than one process is shared, and the rest, which are just referenced by the actions of one process, are the private variables of that process. Hence, actions which do not share any variable can be executed in parallel, and those share a common variable cannot be executed in a same time.

On the other hand, based on the distributed model of parallel execution, the variables of an action systems are partitioned among processes. Consequently, the communication and synchronization between processes is modelled by the shared event model. An event is a shared one, if it refers to variables of two or more processes. As a result, processes can synchronize by executing a shared event and the communication between processes is provided by updating a variable of a process based on the value of a variable in another process, by the occurrence of a shared event. Similar to concurrent action system, events without common variable can be executed in parallel and those with common variables cannot happen in the same time.

Stepwise refinement of action systems has been introduced by Back in [28]. As mentioned before, action systems are a special case of sequential statements. So, Back uses the refinement calculus to develop the stepwise refinement of action system in order to transform a semi-sequential algorithm or an algorithm's specification into an action system which can be executed in a parallel fashion.

To transform the centralized model to a distributed one, as mention before (concurrent and parallel action systems), we need to replace single variable with several variables, which keep the same information but allow a distributed access to it. By this technique, the dependency between actions will be reduced and parallel execution becomes possible.

2.5.1.2 Trace Refinement of Action Systems

Trace refinement of action systems is introduced by Back in [22, 30]. The trace refinement is based on state-traces, where the trace can be infinite.

An action system A defined as

$$z := z_0 \ ; \ \mathit{begin\ var}\ x := x_0 \ ; \ \mathit{do}\ B\ \mathit{od}\ \mathit{end} : z$$

Where x represents the local variables of A, which has been initialized to x_0 and, z represents the global variables and B is a guarded command. A computation of A is either finite sequence

$$(x_0, z_0), (x_1, z_1), \cdots, (x_n, z_n)$$

Where (x_n, z_n) satisfies the exit condition (a successful computation), or it is a finite sequence

$$(x_0, z_0), (x_1, z_1), \cdots, (x_n, z_n), \bot,$$

Where $\bot$ indicate the occurrence of abortion (a failed computation), or it is a infinite sequence

$$(x_0, z_0), (x_1, z_1), (x_2, z_2), \cdots$$

Where no abortion occurs and no exit condition is satisfied all through the sequence. A computation is a *trace* of an action system, if:

- All the local variables (x) are removed,
- All the finite stuttering, caused by the local variables, are removed,
- $\perp$ is left if exists.

The set of traces of an action system A is represented by $tr(A)$.

A *trace specification* of an action system A is a set of sequences of its global variables values, without trailing element $\perp$. System A satisfies a trace specification T if

$$tr(A) \subseteq T$$

An action system A' is a trace refinement of an action system A if

$$\forall\, T \cdot tr(A) \subseteq T \Rightarrow tr(A') \subseteq T.$$

By this kind of refinement the set of different traces of an action system may decrease.

Based on what has been discussed in this section, refinement of reactive systems is a special case of the data refinement with some extra conditions, explained in [22].

2.6 Communicating Sequential Processes (CSP)

Hoare's Communicating Sequential Processes (CSP) [73] is an event-based theory that aims to provide a notation for expressing and reasoning about systems of concurrent processes [103]. Expressing includes designing, specifying and implementing. During reasoning, the system description can be modified and developed in order to verify its correctness. To do that, a formal notation is required, otherwise, it will be difficult to describe a process, precisely enough, to modify it or to contrast it with other possible designs [103].

In this approach, a process communicates with its environment, through atomic events [43]. As a result, its behaviour is defined in terms of the temporal ordering of events.

2.6.1 CSP Algebra

The set of events in which process P can engage is presented by αP and is called its alphabet. Process P behaviour can be specified by $P \mathrel{\widehat{=}} E$, where E is an algebraic

expression [43]. E is constructed from elements of αP, basic processes, and CSP operations. More information about basic processes and CSP operations can be found in [3].

In order to present the sequencing of events the prefix operator ($\rightarrow$) is required. For example, $a \rightarrow P$ express a process that engages in event a and then behaves as process P. Besides, internal and external choices of behaviour can be describe by the choice operators ($[], \sqcap$). An external choice, describes a choice of behaviour towards the environment, whereas, an internal choice as $P \sqcap Q$ represents the process which chooses between behaving as P or Q, internally.

In a parallel composition of processes, composed processes can interact by synchronizing over their common events and other events can occur independently. A common event between two composed processes becomes a single event in their parallel composition and it can be only offered if all the composed processes are ready to offer it. The parallel composition of processes P and Q is expressed as $P \parallel Q$.

It is convenient to hide interaction between composed processes from their environment. This feature is provided by the hiding operator in CSP ($\backslash$). For example, if $C \subseteq \alpha P$, then $P \backslash C$ is a process that behaves as P when all events in C are hidden. How hiding may affect a process is illustrated as follows:

$$\begin{array}{lcll} (a \rightarrow P)\backslash C & = & a \rightarrow (P\backslash C) & if a \notin C \\ (c \rightarrow P)\backslash C & = & P\backslash C & if c \in C \end{array}$$

Hiding an infinite behaviour causes a process to diverge.

2.7 Linking Events and State

A labelled transition-system is a transition system where a label is assigned to each transition, and it may be considered as a CSP process if the labels of transitions are treated as events [43]. Morgan [63] has defined failures-divergences semantics for labelled action systems in terms of weakest-precondition formula.

In [63], Morgan explains that in a typical state-based formalism like action systems, a state is shared between several actions. These actions are either enabled or disabled based on that state. The occurrence of an action changes the state, which cause some changes on the set of enabled events.

On the other hand, in a typical event-base framework like CSP, actions do not have any structure, and do not manipulate any state. As a result, the behaviour of a process is described in terms of sequences of actions.

Combining sate-based and event-based approaches is useful in practice, because there are some aspects of behaviour best described by state, and there are others, best to be described in terms of explicit sequencing.

2.8 Event-B

Event-B [13, 6] is a formal modelling framework, based on set-theory as a modelling notation, use of refinement in order to model a system in different levels of abstraction, and first-order logic to verify consistency of different refinement levels [14]. The fundamental idea is to gradually introduce some simple features during a system design process, that together will eventually result in a global precision of the design.

Usually, in the beginning of modelling, modeller information is incomplete about the system. Event-B helps the modeller to improve his/her understanding in two ways, reasoning about the model, and refinement.

Refinement helps the modeller to handle the complexity by introducing details of a system, gradually, in a rate that ease the understanding. So the model is improved by each refinement until it capture all the important properties.

Besides, reasoning makes it possible for a modeller to verify properties of a model, to analyse a model, and it guides him/her to improve the model.

Our main intention of using formal modelling framework such as Event-B, is to develop a correct system. So the first step is to carefully define the correctness criteria of a system in the *definitions and requirements document*. After producing the definitions and requirements document, there is no guarantee that specified properties of our system can be satisfied. The next step is to *model* the system based on the definitions and requirements document. Modelling is different than programming. In programming we are constructing a formal set of instructions for the computer, to perform some tasks. But our intention in modelling is to formalize a system in which there is a certain piece of software (the final product), as well as its environment. The system (software and its environment) has to be carefully modelled, to understand the exact assumptions in which our final product is going to behave. Based on this methodology the modelling will be the main task of system engineering, and programming will be its sub-task (may be automated). The modelling process in Event-B, is not just about formalizing our mental representation of the future system, but it also includes *proving* that the described properties in the definitions and requirements document, are preserved by that representation.

Since modelling timing properties in Event-B, is the focus of this thesis, we will present an introduction to some of Event-B's main features, in the following sections.

2.8.1 Event-B Core Syntax

Abrial [14] defines Event-B in terms of a few simple concepts to model a discrete event system, and proof obligations to verify properties of that system.

An Event-B model consist of *contexts* and *machines*. A context contains the static parts of a model, and a machine contains the dynamic parts. A machine has a state which is represented in terms of its *variables*. These variables correspond to simple mathematical objects (set, binary relation, numbers, etc). Variables are constrained by *invariants* $I(v)$, where v represents the constrained variables. These constraints have to hold whenever the values of the variables are changed.

Beside state, an Event-B machine has several *events*, which describe how its state may evolve. An event has two parts; *guards* and *actions*. All the guards of an event should hold when that event occurs. The action part of an event specifies how its occurrence changes state variables. Because events are atomic, if the guards of several events hold at the same time, at most one of them could occur at any given time. The order in which those events will be executed is non-deterministic.

It is very important to express the dynamic parts of a system in two ways, in terms of events (state transitions), and in terms of invariants which despite the state changes over time, caused by events' occurrences, the conditions they describe always remain true. During writing the events' actions, modellers do not necessary take into account the invariants. Accordingly, there is no guarantee for the invariants to be preserved by those events, and it has to be proved. So by just expressing a property through some events, there will be no reason for that property to be preserved by them.

An event (`evt`) can be specified in one of the following three forms:

$$
\begin{aligned}
&\texttt{evt} \;\widehat{=}\; \textbf{begin } S(v) \textbf{ end},\\
&\texttt{evt} \;\widehat{=}\; \textbf{when } P(v) \textbf{ then } S(v) \textbf{ end},\\
&\texttt{evt} \;\widehat{=}\; \textbf{any } t \textbf{ where } P(t,v) \textbf{ then } S(t,v) \textbf{ end},
\end{aligned}
$$

Where $P(\cdots)$ represents a predicate specifying the event's guard. $S(\cdots)$ represents the action in which some variables are updated. Also, v denotes the machine variables and the event's parameters are represented by t which are local to the event.

A collection of *assignments* which modifies the state of a machine simultaneously, builds the action section of an event. An assignment may have one of the following forms:

Assignment		**Before-After Predicate**
$x := E(t,v)$	–	$x' = E(t,v)$
$x :\in E(t,v)$	–	$x' \in E(t,v)$
$x :\mid Q(t,v,x')$	–	$Q(t,v,x')$,

Where $E(\cdots)$ represents an expression, $Q(\cdots)$ a predicate, and x some variables. The before-after predicate shows the relation between variables before and after an assignment. In the left hand side x' represent the value of variable x after the assignment. In the following section we will talk about Event-B semantics.

2.8.2 Event-B Semantics

The developers of Event-B claim that it is suitable for diverse modelling domains. Each modelling domain has an appropriate semantics. In order to have semantics appropriate for different models, Event-B semantics is provided implicitly by proof obligations associated with a model. Hallerstede in [67] explains how this approach can be beneficial for modelling. Event-B semantics will be discussed briefly based on Hallerstede's work in the following.

In Event-B, reasoning is considered an essential part of modelling, since it is the necessity to understand complex models, and the meaning of a model arises from what is proved about it. Besides, a systematic support for reasoning is embedded into the Event-B language. What needs to be proved is called a proof obligation of a model in Event-B, which are essential to the method.

Proof obligations verify soundness of a model, in respect to some specified behavioural semantics. They also guide the modeller, since when a proof obligation is failed to be discharged, the proof attempts provide some hints about how the model can be improved. It is not an exaggeration to consider this, as the major importance of proof obligations during modelling process. As a result, modelling in Event-B is substantially based on the interaction of editing a model, and analysing its proof obligations.

In each modelling domain, we want to prove the right facts about the model, and the criterion for the right facts, is a particular behavioural semantics. In Event-B, the sound proof obligations have been evolved to cover semantics of different modelling domains.

2.8.3 Refinement in Event-B

As explained in Abrial's book [13], refinement means to learn and build the model of a system gradually, since people in the stepwise modelling community believe that, practically, it is not possible to build a single model representing once and for all, the reality. In a real world system, the structural complexity (i.e. number and structure of variables and the relationship between them) of its model, makes the modelling and verification processes extremely challenging.

Besides, understanding a single model of the reality is much harder than learning it, step by step, and part by part. In Event-B, the modeller constructs an ordered sequence

of models, where each model refines its preceding model in the sequence. Two types of refinement is imaginable for an Event-B model. These two types will be discussed in the following.

2.8.3.1 Horizontal Refinement (Superposition Refinement)

As mentioned, usually it is not possible to model a large system in one shot, and it needs to be done by a step-wise process. During this gradual improvement, the states and the transitions of a system's components are first created very abstractly and then be enriched by introduction of concrete elements. This process is called *horizontal refinement* [13].

By each refinement, we expose the model to more details, which increase the accuracy of the model. In this process, some parts of the system which were invisible before, will be revealed. In an Event-B model, the revealing of the hidden parts, causes the appearance of new variables.

There is another extension corresponding to this, called *temporal extension* [13]. Temporal extensions add some new transitions, in order to only modify the new variables. This will appear in an Event-B model by the means of *new events*. Since the concrete variables, these new events manipulate, do not exist in the previous abstract levels, the new events refine some implicit events doing nothing (refining `skip`), in those abstract levels. So, a refinement ends up as a discrete observation, which is performing in a finer time granularity in comparison to its abstraction. These new events are similar to auxiliary actions in Back's formalism of action systems explained in section 2.5.1. Same as auxiliary actions which cause finite amount of stuttering, the new events should not diverge. As it will be explained in Section 2.8.4.3, we use variants to guarantee that new events will not be enabled indefinitely.

During horizontal refinement, a modeller goes through the specification and requirements of a system, and gradually chooses some elements from them to be formalized, until there is no property and requirement left. One of the useful outcomes of this process is the traceability of specification and requirements. Often, the modeller finds some incompleteness or inconsistency in the system's specification and requirements during it.

By performing horizontal refinement, it is required to verify that a more concrete refinement step does not invalidate what has been done in a more abstract ones. We will talk about refinement consistency proofs in Section2.8.4.2.

2.8.3.2 Vertical Refinement (Data Refinement)

There is another kind of refinement in which we transform some states and events of the model in order to ease the implementation of the model (based on solution specification). This type of refinement is called *vertical refinement*. Usually, vertical refinements are preformed when all horizontal refinement steps have been performed, but a modeller may decide to have intervals of horizontal and vertical refinements.

A typical example of vertical refinement is to replace finite sets by boolean arrays, which is transformation of a model's data type to another (data refinement).

Same as horizontal refinement, modellers need to perform refinement consistency proofs, in order to verify that the implementation choices are coherent with their abstract view.

In this section the refinement feature in Even-B has been covered. In the following, proof obligations in Event-B will be discussed.

2.8.4 Event-B Proof Method

Event-B is a formal modelling method, so usually the main concern of its users is to learn about the model and to understand it, not be concerned with the technicality of the proving process. To provide a formal modelling environment which satisfies this need, the method has to have a systematic support for generating proof obligations. Then it will be possible to mechanize the process. But this is not the only reason to atomize generating proof obligations. Usually there are thousands of proof obligations associated to a model, and any change in that model can affect many of them. As a result, writing and maintaining them manually, is a time consuming and an error prone process.

In the following, we will present some of the default proof obligations associated with Event-B models, based on [14].

2.8.4.1 Consistency of Machine

An Event-B machine is *consistent* if all of its invariants are preserved by each event of that machine. As a result, for each invariant, it must be proved that if the invariant holds when an event is enabled (its guards hold), the actions of that event, change state variables in such a way, that their new values will satisfy that invariant.

According to what has been explained, one of the standard proof obligation for each event and each invariant is for checking machine consistency. Assume an event *evt* as follows:

evt
any x **where**
$G(s, c, v, x)$
then
$v :| BA(s, c, v, x, v')$
end

Where s denotes the seen sets (in the context), c the seen constants, v the variables of the machine, x represents the event's parameters, and BA is the before-after predicate. In the before-after predicate the primed values is equal to some expression depending on the non-primed value. If $A(s, c)$ represents the axioms and theorems seen by the machine, and $I(s, c, v)$ denotes the invariants and local theorems of it, the consistency PO of an invariant $inv(s, c, v)$ for event evt will be as follows:

Axioms and theorems	$A(s, c)$
Invariants and theorems	$I(s, c, v)$
Guards of the event	$G(s, c, v, x)$
Before-after predicate of the event	$BA(s, c, v, x, v')$
$\vdash$	$\vdash$
Modified specific invariant	$inv(s, c, v')$

The above proof sequence has two parts which are separated by $\vdash$ sign. The left hand side of the sequence is the hypotheses of the proof, and the left hand side is the goal of the proof.

2.8.4.2 Refining a Machine

As explained in Section 2.8.3, a machine may be refined by enriching its states or transitions. The predicates which specify the relation between the concrete states and the abstract ones are called *gluing invariants*.

An event of an abstract machine may be refined by one or more events in the concrete machine. If an event A is refining an event B, then the guards of event A have to be stronger than the guards of event B, and the conjoined action of both events A and B should not violate the gluing invariant.

Some of the default proof obligations in Event-B are responsible to verify these two characteristics of refinements. For a concrete event *con* and its abstract event *abs* as follows:

$$\texttt{abs} \;\widehat{=}\; \textbf{when } P(v) \textbf{ then } v := E(v) \textbf{ end},$$
$$\texttt{con} \;\widehat{=}\; \textbf{when } Q(w) \textbf{ then } w := F(w) \textbf{ end},$$

If *I(v)* represents the abstract invariant and *J(v,w)* is the gluing invariant, the following proof obligation needs to be proved:

$$\begin{array}{l} I(v) \wedge J(v,w) \wedge Q(w) \\ \vdash \\ P(v) \wedge J(E(v), F(w)). \end{array} \tag{2.5}$$

If the abstract and concrete events are parametrized as follows:

$$\texttt{abs} \;\widehat{=}\; \textbf{any}\ t\ \textbf{where}\ P(t,v)\ \textbf{then}\ v := E(t,v)\ \textbf{end},$$
$$\texttt{con} \;\widehat{=}\; \textbf{any}\ u\ \textbf{with}\ t = W(u,w)\ \textbf{where}\ Q(u,w)\ \textbf{then}\ w := F(u,w)\ \textbf{end},$$

then the statement to prove is the following:

$$\begin{array}{l} I(v) \wedge J(v,w) \wedge Q(u,w) \\ \vdash \\ P(W(u,w),v) \wedge J(E(W(u,w),v), F(u,w)). \end{array} \tag{2.6}$$

In the above statement *W(u,w)* is called *witness*. Witnesses specify the relation between abstract and concrete parameters. Witnesses are like local gluing invariants.

In this section, the refinement POs for deterministic assignments have been discussed. Since we just use this type of assignment in this work, POs for non-deterministic assignments have not been mentioned. For further read, the refinement POs for events with non-deterministic assignments, have been discussed in [13].

2.8.4.3 Adding New Events in a Refinement

As mentioned before, in a horizontal refinement, some variables and transitions, which represent the invisible parts of the system in the abstraction, will be revealed in the concrete model. Such transitions have to be proved to refine a dummy event (*skip*), which does nothing in the abstract machine.

Besides, it may be proved that it is not possible for the new events to take the control forever. The reason is that the behaviour of refining machine should include the abstract one. If the abstract events do not have the chance to have the control, then the concrete machine does not have the abstract machine behaviour. In order to verify this, a *variant expression* $V(w)$ has to be provided which is decreased by each occurrence of a new event. So, it will be guaranteed that the new events cannot occur infinitely. The associated proof obligation is called *convergence*. For a new event *evt*

$$\texttt{evt} \;\widehat{=}\; \textbf{where}\ R(w)\ \textbf{then}\ w := G(w)\ \textbf{end},$$

the required proof obligations to be proved, are as follows:

$$I(v) \wedge J(v,w) \quad \vdash \quad J(v, G(w)), \tag{2.7}$$

$$I(v) \wedge J(v,w) \quad \vdash \quad V(w) \in \mathbb{N} \wedge V(G(w)) < V(w). \tag{2.8}$$

In the above statements, the *variant expression* is assumed to be a natural number for simplicity, but it can be more complex. Proof obligation 2.7 checks whether the actions of the new event do nothing in the abstract machine, by proving that its corresponding gluing invariant will hold, after the new event's occurrence. Proof obligation 2.8 checks if the new event can only occur finite number of time, because by each occurrence of it the corresponding variant expression will be reduced.

Variant expression can be express in terms of a finite set. In this case proof obligation 2.8 will be changed as follows:

$$I(v) \wedge J(v,w) \quad \vdash \quad V(G(w)) \subset V(w). \tag{2.9}$$

So by each occurrence of the event, the set will shrinks, and since it is finite, the eventual disabling of the new event is guaranteed.

2.8.4.4 Deadlock Freedom

By guarding the events of an Event-B model, it is possible to reach a state where none of the guards are true (deadlock). Sometimes, this a desirable behaviour of the system, but if one of the desired system properties is deadlock freedom, then it should be verified that there will be an enabled event all through the life cycle of the system.

In DFL (2.10), for a model with constants c, set of axioms $A(c)$, and set of invariants $I(c,v)$, we prove that the guard of at least one of the events, $G_1(c,v), \cdots, G_m(c,v)$, always hold.

$$\begin{array}{l} A(c) \\ I(c,v) \\ \vdash \\ G_1(c,v) \vee \cdots \vee G_m(c,v) \end{array} \tag{2.10}$$

By proving DLF for a machine, we prove that there is some enabled event in every reachable state of that model. This proof obligation will not be generated by the Rodin tool-set automatically, since for a complex model it will be very complex and challenging to be discharged.

Some of the main standard proof obligations of Event-B proof method have been introduced in this section. In the next section, the model decomposition in Event-B will be discussed.

2.8.5 Decomposition

Modelling a large system, will end up with a complex model with unmanageable number of states and transitions. Proofs in large models are more difficult to do. One of the approaches to deal with this matter is *decomposition*. Abrial [16] and Butler [40] have introduced two possible approaches to decompose an Event-B model of a system to its various components. We will go through these two approaches in this section based on their works.

As explained in Section 2.8.3, although a modeller starts with a simple model of a system, containing a few numbers of state variables and events, but by each refinement, new state variables and events will be added to the model. This process usually causes the model to end up with so many events and state variables that performing refinement becomes difficult to manage. Besides, as the model contains more details of the system specification, refinements may not involve the entire system any more, and they are just concern a few variables and events. In this situation, decomposition comes to help the modeller, by breaking a large model into smaller pieces. These smaller pieces can be refined independently which makes the process more scalable.

Decomposition has to be done in a way that independent pieces, can always be re-composed easily. Besides, the re-composed model has to be guaranteed, to be a refinement of the original model. As a result, decomposition is a kind of divide-and-conquer approach, to solve complex problems.

An Event-B model can be decomposed either by event-based synchronization or state-based synchronization.

2.8.5.1 Shared-Variable Decomposition

In this approach, the independent pieces of a decomposed model, interact based on a shared state (represented by one or several state-variables). Imagine an Event-B machine M with four events, $e1, e2, e3$ and $e4$, which we want to decompose to two machines, $M1$ and $M2$. In order to do that, it is required to split the state-variables too. Suppose $v1, v2$ and $v3$ are the state-variables of M, where $v1$ is involved in $e1$ and $e2$, $v2$ in $e2$ and $e3$, and $v3$ in $e3$ and $e4$.

In order to split the variables, $v1$ will goes to $M1$ and $v3$ goes to $M2$, but $v2$ is a *shared variable* and cannot be split. The only way to decompose the model, is to replicate the

shared variable. The replicated shared variables in each component, are referred to as *external variables*.

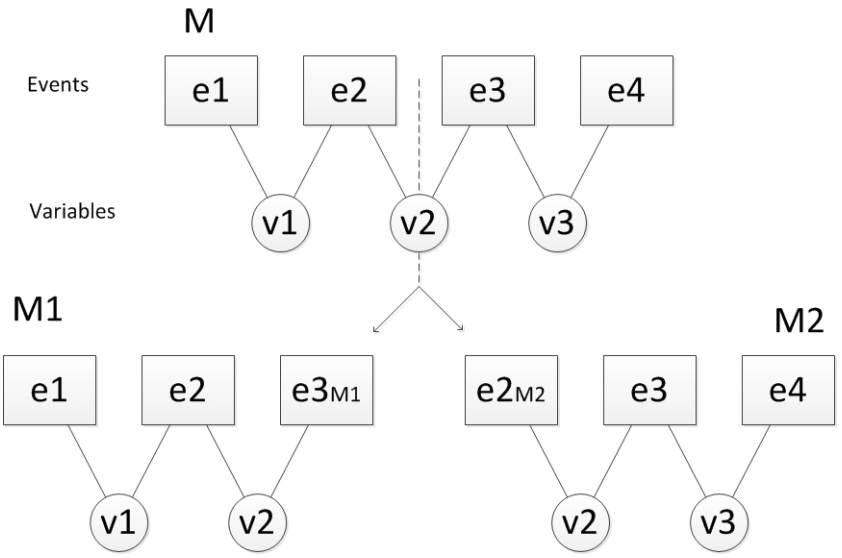


Figure 2.1: Shared variable decomposition of an Event-B Machine

Since they can be refined independently, it is possible that the shared variable be data-refined in different manners in each component, which will end up with components that cannot be re-composed. To solve this problem, the shared variable should not be date refined.

But this is not enough, since in a component that just reads the shared variable and does not modify it, shared variable becomes a constant. To solve this problem, we need some events, in each component, in which the use of shared variable before decomposition is mimicked (simulate how shared variables are modified in other decomposed components). These events are referred to as *external events*. In our example, suppose event $e2$ with guard $G_2(v1, v2)$ and before-after predicates $E_2(v1, v1', v2, v2')$. Then event $e2_{M2}$ in machine $M2$, is an external event if the following predicate can be proved:

$$\begin{aligned} & G_2(v1, v2) \ \wedge \ E_2(v1, v1', v2, v2') \\ & \Rightarrow \\ & G_{2_{M1}}(v2) \ \wedge \ E_{2_{M1}}(v2, v2'). \end{aligned} \tag{2.11}$$

Since an external event in a component, mimics how the shared variables are modified in other components, it cannot be refined.

2.8.5.2 Shared-Event Decomposition

Butler [40] has introduced the shared-event decomposition of Event-B models, inspired by the synchronous parallel composition of processes, which can be found in process algebra such as CSP.

A parallel composition of machines M and N, is represented as $M \parallel N$. Based on shared-event decomposition, M and N should have no common state variable, and the synchronization must happen through shared events. For machines M and N with shared events $evt1$ and $evt2$

$$\begin{aligned} evt1 &= \textbf{any } y \textbf{ where } G(y,m) \textbf{ then } S(y,m) \textbf{ end} \\ evt2 &= \textbf{any } z \textbf{ where } H(z,n) \textbf{ then } T(z,n) \textbf{ end}, \end{aligned}$$

Where m is the state variable of M and n is the state variable of N, to achieve the synchronization effect between them, events $evt1$ and $evt2$ will be fused by using the parallel operator for events,

$$\begin{aligned} evt1 \parallel evt2 \;\widehat{=}\; & \textbf{any } y, z \textbf{ where} \\ & G(y,m) \;\wedge\; H(z,n) \\ & \textbf{then} \\ & S(y,m) \parallel T(z,n) \\ & \textbf{end}. \end{aligned}$$

The parallel operator represents a simultaneous occurrence of the shared events' actions in the composed event, only when the guards of both events hold. As a result, the synchronization between M and N happens when the composite system engages in event $evt1 \parallel evt2$, which can only happen if both machines are willing to engage in it.

To provide a better understanding of the shared-event decomposition, we will go through a simple example. Consider an Event-B machine A with three events $e1$, $e2$ and $e3$, and two state variables $v1$ and $v2$, as shown in Figure 2.2.

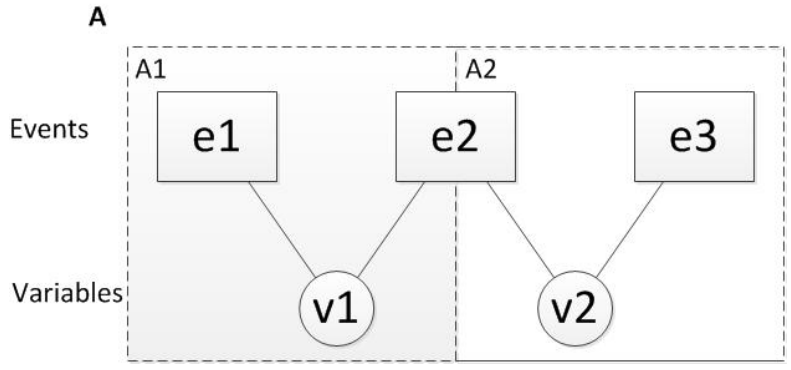


Figure 2.2: Decomposing machine A into A1 and A2

Variable $v1$ is used in events $e1$ and $e2$, and variable $v2$ in events $e2$ and $e3$. So, if A is decomposed in two sub-machines, $A1$ and $A2$, where $v1$ goes to $A1$, and $v2$ goes to $A2$, then event $e2$ will be the share-event, since it uses both of those variables. After decomposition of A into $A1$ and $A2$, $e2$ will appear in both of them, but its guards and actions related to $v1$ will just appear in $A1$, and those related to $v2$ will appear in $A2$.

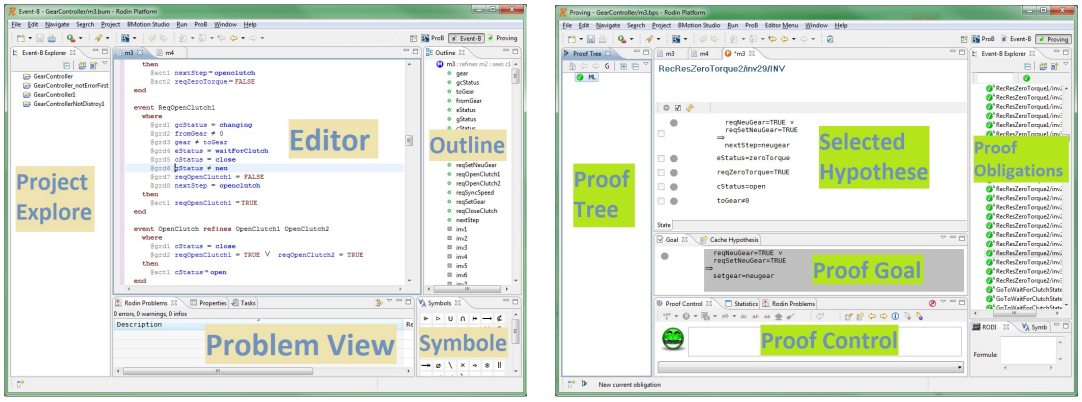
The biggest advantage of shared-event approach is that it is possible to refine shared-events independently in each component. Besides, since there is no shared variable, there is no restriction on data refinements either.

There is a tool support for both types of decomposition in the Rodin tool-set [107], which has been used in this work.

So far, Event-B features related to this work, such as refinement, decomposition and proof obligations, have been covered. The following section will talk about Rodin tool-set.

2.9 Rodin Tool-set

Rodin [8] is an Eclipse-based [5] integrated development environment (IDE) for Event-B, which supports refinement and mathematical proof. The Event-B language is developed with extendibility in mind, and since Rodin is Eclipse-based, it can be easily extended for different problem domains by *Plug-in* development.



(a) Modelling user interface perspective

(b) Proving user interface perspective

Figure 2.3: The default user interfaces of the Rodin tool-set.

There are two interfaces available by default, in the Rodin tool-set, the modelling user interface (MUI), and the proving user interface (PUI). These two interfaces have been developed by extending the Eclipse perspectives, and they are strongly integrated, since reasoning is considered as a part of modelling process in Event-B method. The screen shots of both interfaces are shown in Figure 2.3.

PUI can be used to do interactive proofs. Sometimes some of the POs of a model cannot be discharged by the automatic prover, either because it is not possible to imply the proof's goal from its hypothesis, or a time-out occurrence. To discharge a PO, the automatic prover applies a set of tactics to the the proof's goal and the selected hypothesis in order to prove that either there is a contradiction in hypothesis, or the goal appears in the hypothesis, or the goal predicate has a true value. But if the model is complex, all the relevant hypothesis may not be selected, or it is not possible to find a correct combination of proof tactics, in order to discharge the PO in a reasonable period of time. As a result, a time-out has been defined in which the proving process of a PO will be aborted, to continue the proof for the next one. In this way the automatic prover can go through all the POs in a reasonable period of time, and then the modeller can investigate the undischarged POs, one by one.

The proof manager which is responsible to maintain proofs status and proofs associated with the POs in the Rodin tool-set, builds a partial/complete proof for a PO by constructing proof trees [14]. Proof trees are recursive structures, consisting of proof tree nodes. A proof node is built of three components, a sequent, a proof rule and a list of child nodes. Proof rules [14] in their pure mathematical form are tools to perform formal proof. A proof tree node is either pending if no proof rule has been applied to it, or non-pending otherwise. An important property of the proof tree is that both the proof manager and the modeller can calculate the proof dependency based on the hypotheses and goals at each node of the proof tree.

The automatic prover creates an initial pending node for each PO, and then applies some predefined automatic tactics in order to discharge it. Tactics [14] provide convenient ways of constructing and manipulating proofs. A tactic's input is a proof tree node, and its output is a boolean which shows whether the tactic modified the node successfully or not.

In the PUI, the modeller is able to add new hypotheses and apply tactics on the proof's goal and hypothesis. If a new hypothesis has been added by the modeller, its correctness has to be proved. Besides, the modeller can manipulate a proof tree by removing nodes.

In this work, the Rodin tool-set has been extended in order to support modelling timing properties based on our approaches. How it can help modeller to add timing properties will be discussed in Chapter 10. In the next section we will look at refinement diagrams.

2.10 Event Refinement Diagrams

The *Event Refinement Diagram* notation has been introduced by Butler [40]. It has been inspired by the structural diagrams of *Jackson System Development (JSD)* [78]. The aim of this notation is to help the modeller, in structuring of the refinements, in

which, new events are introduced to decompose the atomicity of some abstract events, into the smaller concrete sub-atomic steps (atomicity decomposition).

In this report there will be several refinement diagrams. It was a helpful tool, while applying our approaches on different case studies. It helped us to handle the complexity of the refinement relations between abstract and concrete machines.

During a stepwise modelling process of a system, the modeller needs to be concerned about the consistency of the model. This involves a lot of invariants, variables and guards. The refinement diagrams are helpful to trace the relation of the concrete and the abstract events (the abstraction of each concrete event, and the relation of skip and abstract events).

The refinement diagram has a tree form structure. An abstract event is positioned on the top of a diagram as its root, and its concrete events or the new events, which model the pre or post steps of the abstract event, are located at the bottom of the structure, as the leaves of the tree. Besides, the concrete events are ordered from left to right based on their occurrence order.

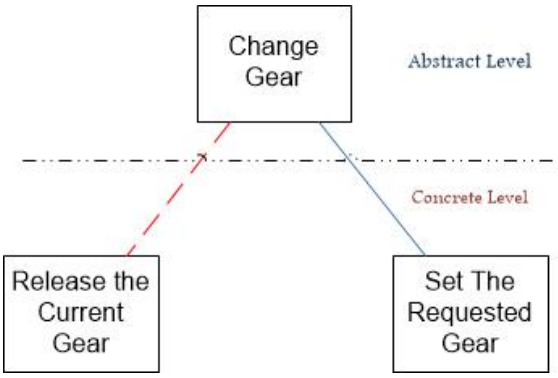


Figure 2.4: Refinement diagram example

As mentioned in Section 2.8.3, in a refinement, some new events may be introduced which are invisible in the abstract machine. Although they are not refining any abstract event, but they represent the required pre and post steps of some abstract events. Other events extend or refine the abstract events.

In a refinement diagram, the relation of an abstract event with the newly introduced events in the concrete machine, is distinguished from its relation with the events, refining it. The first group must be connected to their corresponding abstract event by dash lines, whereas, the refining events are associated to their abstract event by solid lines.

As shown in Figure 2.4, event *ReleaseCurrentGear* is a new event in the concrete machine, which represents the required pre step of changing the engaged gear, in a car. So it is connected by dashed line to the abstract event. On the other hand, event

SetRequestedGear refines the abstract event, and it is connected by a solid line to its abstract event.

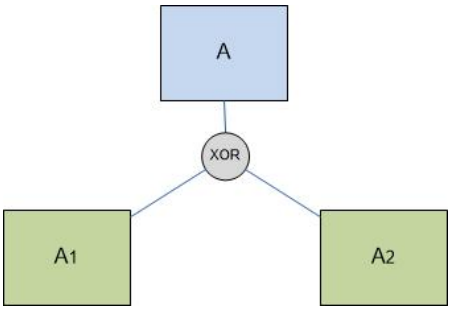


Figure 2.5: Application of XOR in a refinement diagram.

The other construct used in this report is *XOR*. Refinement of an event by several alternative events in the concrete machine can be presented by using *XOR*. In the example of Figure 2.5, the refinement diagram represents a refinement where either of occurrences of the concrete events (A_1 and A_2) will be equivalent to the abstract event's occurrence (A).

The final construct to be explained is the loop construct. By using it, it is possible to present finite or infinite loops of events in a refinement diagram. In Figure 2.6 the abstract event has been refined by an iterative concrete event. The star in the diagram represents an infinite loop.

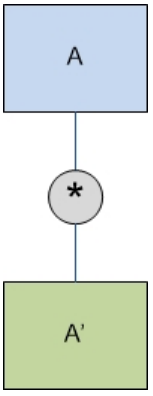


Figure 2.6: Expressing a loop in a refinement diagram.

In this section refinement diagram and its constructed, used in this report, have been introduced and briefly explained.

In this chapter, the concepts required for following the discussion of modelling real-time systems in Event-B, have been explained. The next chapter will provide a brief overview of what real-time systems are, as well as existing works on modelling them and reasoning about them.

Chapter 3

Timed Verification and Reasoning

Since the focus of this study is modelling and verifying of the timing properties of real-time systems, in this chapter a range of real-time related works will be investigated.

In Section 3.1 some general information about real-time systems is given. Section 3.2 talks about model checking approach to verify real-time systems. Besides, some existing model checking methods and tools will be discussed briefly. Section 3.3 talks about the *Timed Automata* approach, used in model checking of real-time systems. Section 3.4, briefly introduces Lamport approach to model the physical continuity of real-time systems, in terms of discrete events. Sections 3.5, 3.6, and 3.8 talk about the real-time extensions of CSP and action systems. Section 3.9 introduces Real-time VDM, an state-based modelling language for real-time systems, followed by Section 3.10, which talks about combining VDM and Co-Simulation to model continuous processes. Section 3.11 talks about some works on modelling timing properties in classical B, which have influenced our work. In the end, in Section 3.12 some of the existing works on modelling timing properties in Event-B will be mentioned.

3.1 Real-time Systems

Before going through the discussion of Real-time Systems modelling and verification, it is essential to have some definitions about time related concepts. These definitions have been elicited from Kopetz book on real-time systems [81].

Assume the time flow as a direct time line, from past to future, based on Newtonian model of time [9]. Then an *instant* is a cut of a time line. The present point in time, is called *now*, which separates the past from the future. A *duration* is an interval on a time line which is defined by a *start event* and a *terminating event* of the interval. A discrete clock partitions the time line into *granules* of clock. *Granules* are sequence of equal

spaced durations, and the *tick* event of a clock is a periodic event, which determines the granules of the clock.

A real-time system, changes as a function of time, where the correctness of a system behaviour depends not only on its sequence of events, but also on when these events occurred. In a distributed real-time system, different components interact through a real-time communication network. As a result, what distinguish real-time systems, is the precise emphasis on their temporal specifications.

Real-time systems have to react to their environments' changes within the time intervals forced by their environments. A *deadline* is a instant by when a result must be produced in a real-time system. There are two kinds of deadline, if a result is useless, if its deadline is passed, its deadline is classified as hard, otherwise it is soft. A system with at least one hard deadline is called a *hard real-time system* or a *safety-critical real-time system*. As a result, a hard real-time system has to guarantee a specific temporal behaviour under all the specified states. On the other hand, a soft real-time system may miss a deadline, one in a while. An example of soft real-time system is an airline reservation system. In this system, if the system cannot keep up with the demands, the response time will be extended and it will just cause the users to slow down. But in a hard real-time system such as a pressure controller of a boiler, missing a deadline can cause an explosion.

There are others classification of real-time systems. We will briefly go through some of them in the following. The first categorization is based on a system behaviour in a failure state. A system failure is a condition that causes a system to fail in performing its required functionalities, when it is required to perform it [35]. Many hard real-time systems have some safe states, which can be reached in case of a system failure. If a real-time system can identify and quickly reach such a safe state, then it will be classified as a *fail-safe* real-time system. But, there are real-time systems which can not identify safe-states. In the case of failure, these systems remain operational by providing a minimal level of service. As a result, they are categorized as *fail-operational*. An example if fail-safe real-time system is railway signalling system, in which if a failure is detected, all the signals can be set to red an stop all the trains in order to bring the system to a safe state. But the flight control system of an airplane must always remain operational and provide a minimal level of service to avoid a crash.

The other categorization of real-time system is based on the stimuli of a real-time system. Real-time systems are classified based on the type of the triggers of their internal behaviour. So it is not about the external behaviour of a real-time system. An event that causes the start of some actions in a system is called *trigger*. As a result, based on the triggering mechanism for the start of communications and processing actions, real-time systems can be classified into *event-triggered* or *time-triggered* systems. In an event-triggered system, all the communications and processes are triggered by events' occurrences other than the *tick* event. Whereas, in a time-triggered one, all activities

are initiated by the progression of real-time. An example of an event-triggered system, is when a button such as call button of an elevator is implemented by an interrupt event in the controller system. On the other hand, if the button is implemented by being sensed periodically, then it will be a timed-triggered request, since a button push will be recognized by the next iteration of the sensing event.

As mentioned, real-time systems may change as a function of time. In a model, based on continuous time, the domain of this function is continuous. As a result, it is possible to specify the state of the model for any given time (Real number). On the other hand, it is possible to model a system based on discrete time, where the domain of the time's function, is provided by a finite iterative sampling of the time line. Our focus in this thesis will be modelling and verifying discrete timing properties.

3.2 Model Checking

Model Checking [47] is a computational method for verifying systems' properties, introduced by Clarke and Emerson [48, 46] for automatic verification of the reactive systems, modelled in terms of finite state-machines. Based on their method, system specification is expressed in a propositional temporal logic and the system is expressed as a state transition model. To verify a system, whether or not its model ($\mathcal{M}$) satisfies its specification (ϕ) must be computed ($\mathcal{M} \models \phi$) [76].

Temporal logic can be categorized in two groups based on its particular view of time, *Linear-time Logic (LTL)* and *Computation Tree Logic (CTL)*. LTL treats time as a set of paths, where each path is a sequence of time instances, whereas CTL has a tree form structure to present time, where the present is the root and future is branching out of it [76].

To have a better understanding of temporal logic the syntax of CTL, and LTL will be explained briefly. The minimal syntax of CTL [76] is as follows:

$$\phi ::= \bot \mid \top \mid p \mid (\neg\phi) \mid (\phi \wedge \phi) \mid (\phi \vee \phi) \mid (\phi \rightarrow \phi) \mid AX\phi \mid EX\phi \mid \quad (3.1)$$

$$AF\phi \mid EF\phi \mid AG\phi \mid EG\phi \mid A[\phi\ U\ \phi] \mid E[\phi\ U\ \phi] \quad (3.2)$$

Where p ranges over a set of atomic formulas, A means along all paths (inevitably), E means along at least one path (possibly), X means next sate, F means some future state, G means all future states (globally), and finally U means until.

Based on CTL, the examples of temporal properties in Section 2.3 can be expressed as follows:

- For all state of the system, it is never the case that P_1 and P_2 use the shared resource x in a same time,

$$\mathbf{AG}\ \neg(P_{1_{u_x}}\ \wedge P_{2_{u_x}})$$

- Whenever P_1 wishes to use the shared resource, it will eventually do so,

$$\mathbf{AG}\ (P_{1_{w_x}}\ \rightarrow\ \mathbf{AF}(P_{1_{u_x}}))$$

- If there is a sequence where in position $j > 0$, P_2 is waiting to have access to shared resource, there is a position $s > j$ in that sequence where P_2 is using the shared variable (Another way of expressing the previous property for P_2).

$$\mathbf{AG}\ (P_{2_{w_x}}\ \rightarrow\ \mathbf{AF}(P_{2_{u_x}}))$$

Where $P_{i_{w_x}}$ means process P_i wishes to use resource x, and $P_{i_{u_x}}$ means process P_i is using resource x.

The LTL formulas are built from predicates with the usual propositional connectives, $\vee, \wedge, \Rightarrow, \neg$ plus two temporal operators, $\circ$ and $\mathbf{U}$ [109]. Operator $\circ$ is read as next, so LTL formula $\circ\varphi$ means φ is satisfied at the next time instant. On the other hand, the operator $\mathbf{U}$ is read as until and $\phi\mathbf{U}\varphi$ means that formula ϕ is satisfied until formula φ is satisfied.

By using the until operator two applicable operators, $\Diamond$ (eventually) and $\Box$ (always), can be defined as follows:

$$\Diamond\varphi \;=\; true\mathbf{U}\varphi \tag{3.3}$$

$$\Box\varphi \;=\; \neg\Diamond\neg\varphi \tag{3.4}$$

$\Box\varphi$ means that formula φ holds for all future times, whereas $\Diamond\varphi$ means formula φ will hold at some time in the future.

So far, temporal logic and its categorization based on its model of time have been explained. In the following, one of the important formalisms in real-time model checking, will be discussed.

3.3 Timed Automata

Timed automata [21] is a formalism for modelling real-time systems, which annotates state-transition systems, with timing constraints, related to a finite set of clocks. In timed automata, each state is related to clocks [18]. A timed automaton TA is a six-tuple (Σ, S, S_0, S_F, X, $\triangle$) where [95]:

- Σ is a finite set of events,
- S is a finite set of states,
- S_0 is a finite set of initial states where $S_0 \subseteq S$,
- S_F is a finite set of accepting states where $S_F \subseteq S$,
- X is a finite set of clocks,
- $\triangle$ is a finite set of transitions where $\triangle \subseteq S \times S \times \sum \times \Phi_X \times 2^X$.

As shown above, in timed automata, each transition contains five parts where Φ_X is the timing constraints on that transition. For example transition $T = (s, s', a, \phi, R)$ causes a jump from state s to state s', when the specified timing constraints ϕ on clocks R, are met, and event $a \in \sum$ occurs. After occurrence of this transition, the clocks in R will be reset. In this example it can be seen how a real-time system could be modelled by using timed automata formalism, where transitions may have timing constraints on some declared clock variables.

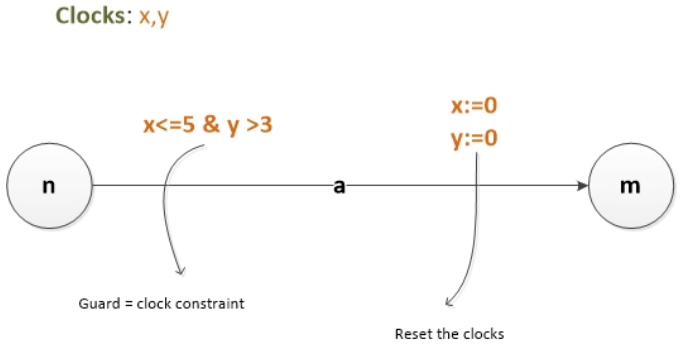


Figure 3.1: An example of a timed automaton

To have a better understanding of the formalism, let us assume a transition of a timed automaton $\mathcal{A} \mathrel{\widehat{=}} (\Sigma, S, S_0, S_F, X, \triangle)$, presented in Figure 3.1. Based on what has been explained so far, the presented transition can be formalized as follows:

$$T \quad = \quad (n, m, a, x \geq 5 \wedge y > 3, \{x, y\})$$

Where T belongs to $\triangle$, n, m are members of S, x, y belong to X, and a is a member of Σ.

In timed automata, a *time sequence* $\tau = \tau_1 \tau_2 \cdots$ is an infinite sequence of time values $\tau_i \in \mathbb{R}$ with $\tau_i > 0$ where

- Monotonicity: $\forall i \cdot i \geq 1 \Rightarrow \tau_i < \tau_{i+1}$,
- Progress: $\forall t \cdot \exists i \cdot t \in \mathbb{R} \wedge i \geq 1 \wedge \tau_i > t$ [21].

A *timed word* [21] over an alphabet $\sum$, is a pair (σ, τ) where $\sigma = \sigma_1\sigma_2 \cdots$ is an infinite word over $\sum$ and τ is a time sequence. Based on these definitions, a *timed language* [21] over alphabet $\sum$, will be a set of timed words over it.

By considering a timed word (σ, τ) as an input to an automata, each member of σ will be interpreted as an event occurrence, and its corresponding component in τ will be its occurrence time.

For a better understanding of timed automata, based on these definitions, an example of a timing property, is given in the following. Assume the alphabet $\{a, b\}$ where no b may happen after time 5. This property can be modelled by defining a timed language L_1 such that:

$$L_1 = \{(\sigma, \tau) \mid \forall\ i \cdot ((\tau_i > 5) \Rightarrow (\sigma_i = a))\} \tag{3.5}$$

Based on (3.5), there is no timed word in L_1 where b has occurred sometime after 5.

In the following, some of the popular real-time model checkers, which use timed automata, will be introduced briefly.

3.3.1 UPPAAL

UPPAAL [85, 34] is a product of a cooperation between University of Uppsala and University of Aalborg. It has been developed to model, simulate and verify real-time systems that can be modelled as a collection of processes, which have finite controlling states, real-valued clocks. These processes communicate through channels or shared variables.

This framework has three main features, required for verification of a real-time system. The first one is a modelling environment based on a non-deterministic guarded-command language, facilitated by real-valued clock variables and simple data types, where a real-time system can be modelled as networks of timed automata, and data variables. Second feature is a simulator to examine dynamic behaviour of a model in its early stage of design by the user. Finally, UPPAAL has a model checker to validate the specification of a system, in the model, by automatically checking invariants and bounded liveness properties.

In order to model the timing properties of a system, state transitions can be guarded based on the values of the clocks. Besides, a transition from a state, can be forced to happen within a duration, by declaring a timing invariant on that state.

UPPAAL is able to check invariants and reachability properties by exploring the system state-space. Its state explorer is designed to have efficient algorithms and data structures. One of the advantages of the UPPAAL model checker, is its ability to provide diagnostic traces for invariants that have not been satisfied.

The properties specification language of UPPAAL, is a restricted subset of *timed computation tree logic (TCTL)* [20], which provides the required notation to express the safety, the liveness, the deadlock, and the response properties of a system [69]. The following temporal operators have been supported in UPPAAL specification language:

- E : exists a path,
- A : for all paths,
- [] : all states in a path,
- <> : some state in a path,

By using these operators the following queries can be declared in the UPPAAL's simulator to be checked:

- $A[]p, A <> p, E <> p, E[]p$, and $p \longrightarrow q$ (followed by)
  where p and q are local properties

A local property p can be declared as follows:

$$p ::= a.l \mid gd \mid gc \mid p \; and \; p \mid p \; or \; p \mid not \; p \mid p \; imply \; p \mid (p) \tag{3.6}$$

Where a represents a process name, l represents an automaton location (state of a process), gd represent a data guard, and gc represents a clock guard.

UPPAAL provides parallel composition, by modelling a system as a collection of processes, and provides some synchronization mechanism for them. But, as explained before, modelling a large system as a single step process does not seems to be practical. So, supporting hierarchical modelling structure is important for a useful method in real world practises. Hierarchical timed automata (HTA) formalism has been introduced by David and Moller [49] as a hierarchical real-time formalism, which enforces some strong well-formedness constraints on UPPAAL syntax to guarantee the consistency of the hierarchy.

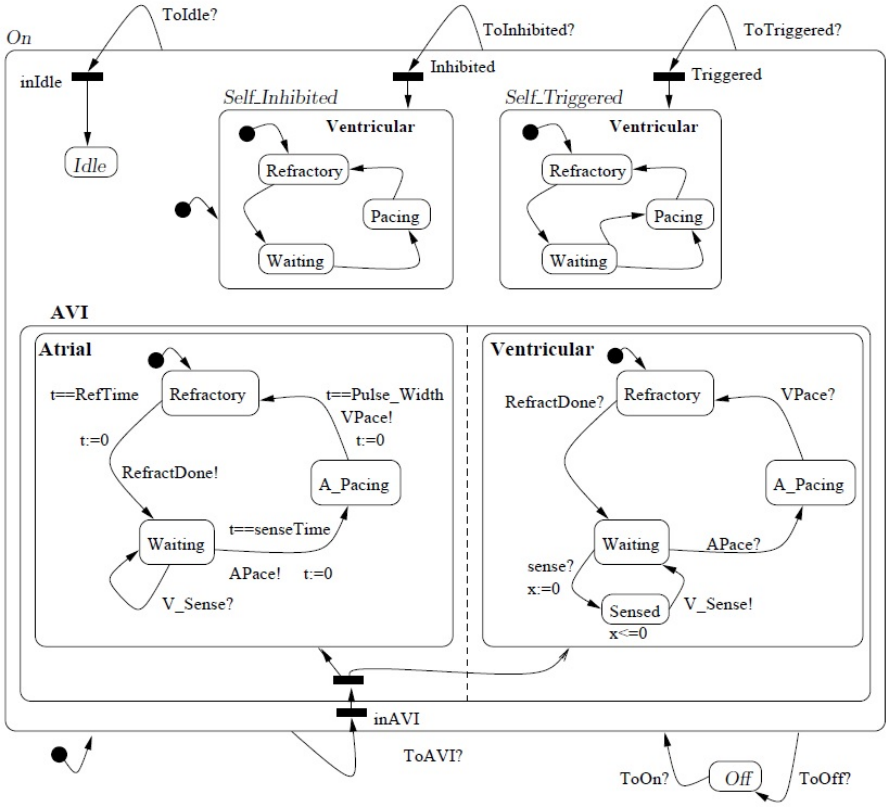


Figure 3.2: A cardiac pacemaker HTA model. In this model off is a basic location and On is the superstate.

Hierarchical timed automata are hierarchical state machines where basic units of control are called locations, which are either basic states or superstates. The relation between a superstate and its sub-states can be a XOR where if a super state is active one and only one of its sub-states is active, or it can be an AND relation where activation of a super states is equivalent to activation of all of its sub-states. There is no tool support for HTA, but some approaches have been introduced in [49] to transform a HTA to a flat UPPAAL model, but this process needs to be done manually.

A hierarchical timed automaton [49] is a tuple $\langle S, S_0, \delta, \sigma, V, C, Ch, T \rangle$ where

- S is a finite set of locations,
- $S_0 \subset S$ is a set of initial locations,
- $\delta : S \to 2^S$ maps abstract locations to their concrete locations,
- $\sigma : S \to \{AND, XOR, BASIC, ENTRY, EXIT, HISTORY\}$ is a type function on locations,
- V, C, Ch are sets of variables, clocks, and channels.
- T is the set of transitions.

As shown δ and σ have been added to provide the means of constructing hierarchical structures. Since hierarchical timed automaton are not the focus of this work we will not go into the details of locations' types. An example of a hierarchical timed automaton has been presented in Figure 3.2, in order to give a better understanding of how states can be evolved in this formalism. The example has been elicited from [49].

3.3.2 KRONOS

KRONOS [117] is a tool for formal model checking of real-time systems based on timed automata and temporal logic. Similar to UPPAAL it has an integrated verification engine with its modelling environment, and embodies a shared clock, which timing constraints of a model are based on its value (e.g., execution times, deadlines, propagation delay).

A real-time system may be modelled in KRONOS, as a composition of timed automata executing in parallel. To model inter-process communications, its transitions are labelled by sets of identifiers. Those identifiers are interpreted as synchronization channels.

The specification framework of KRONOS, let us to define the correctness criteria of a system in two different ways. It can be declared as formulas of the timed computation tree Logic (TCTL) [20], which is a type of timed temporal logic [19, 68]. This is a logical approach and to verify the satisfaction of those formulas, KRONOS applies some model checking algorithm. The other approach is to declare the properties of a system in terms of timed automata, which is a behavioural approach.

3.3.3 Real-time Promela

Real-time Promela (Process Meta Language) [110] is an extension of the Promela language [75, 102] to support timing information and clocks. Promela is a specification language for interactive concurrent systems, consists of a finite number of components. These components act independently and communicate through shared variables or message channels. Message channels may be defined synchronous or asynchronous.

A Promela model consists of type declarations, channel declarations, variable declarations, and process declaration (including the initialization process). Each processes is defined based on a type, defined by a *protype*, which is similar to a class in an object-oriented language. All the processes are executed concurrently. A process can be created at any time, within any process. The body of a process consists of a sequence of atomic statements. At any given point, a process may have several enabled executable statements, which will be scheduled non-deterministically to be executed.

Two types of properties can be specified by Promela, safety and liveness properties. Liveness properties are specified by using LTL formulas, and safety properties are defines in terms of invariants.

SPIN (Simple Promela Interpreter) [74] is a tool-set for analysing the consistency of concurrent systems, which are described in Promela specification language. The safety properties in SPIN are checked by trying to find a trace leading to a violation of the properties. If none has been found the properties have been satisfied by the system specification. On the other hand, a liveness property is checked by looking for an infinite loop which does not satisfy the property. If none has been found, then the liveness property is satisfied.

Tool support is provided for real-time Promela by extending the SPIN tool-set. In the extended SPIN, in addition to what can be expressed in standard Promela, it is possible to constrain a statement, based on the possible values of a clock, or the relative values of two clocks. But, by adding time, the size of the state space is significantly increased in most of the cases.

In this section, some of the real-time model checkers, based on timed automata have been introduced, and whether they support some of the features we are interested in, such as refinement and decomposition have been discussed.

3.4 An Old-Fashioned Recipe for Real-time by Lamport

An Old-Fashioned Recipe for Real-time [12] has been introduced by Lamport to model real-time systems by using the traditional methods of specifying and reasoning about concurrent systems. In his approach, the *temporal logic of actions* (TLA) [83] is used to express the untimed version of a real-time system, joined by its timing properties, expressed in terms of parametrized predicates.

In TLA, temporal logic and the logic of actions have been combined, in order to provide a formal framework for specifying and reasoning about concurrent systems. In TLA, systems and their properties are represented in the same logic. To express formulas in TLA, other than mathematical operators (such as $\wedge$), three new operators have been introduced: $'$ (prime), $\Box$ (read always), $\Diamond$ (read eventually), and the hidden operator

(used to exclude some variables from a property). TLA syntax is as follows:

$$
\begin{aligned}
\langle formula \rangle &\triangleq \langle predicate \rangle \mid \Box[\langle action \rangle]_{\langle state\ function \rangle} \mid \neg\langle formula \rangle \\
&\quad \mid \langle formula \rangle \ \wedge \ \langle formula \rangle \mid \Box\langle formula \rangle \\
\langle action \rangle &\triangleq \text{boolean-valued expression containing constant symbols,} \\
&\quad \text{variables, and primed variables} \\
\langle predicate \rangle &\triangleq \langle action \rangle \text{ with no primed variables} \mid Enabled\langle action \rangle \\
\langle state\ function \rangle &\triangleq \text{nonboolean expression containing constant symbols} \\
&\quad \text{and variables}
\end{aligned}
$$

An atomic operation in a concurrent program is expressed in terms of action. An action is an expression consisting of variables, primed variables, and constant symbols, where unprimed variables represent the old state and the primed ones refer to the new state.

One of the main feature of TLA is its support for fairness requirements. Lamport has introduced two types of fairness, strong and weak

$$strong\ fariness: \quad \Box((\Diamond\ executed)\ \vee (\Diamond\ impossible)) \tag{3.7}$$

$$weak\ fariness: \quad \Box((\Diamond\ executed)\ \vee (\Diamond\Box\ impossible)) \tag{3.8}$$

Based on strong fairness, an operation must be executed if it is often enough possible to do so, where often enough means infinitely often. On the other hand, weak fairness asserts that an operation must be executed if it remains possible to do so, for a long enough time, where long enough means until the operation is executed.

Lamport's approach is based on the idea, that the physical continuity of real-time systems can be modelled in terms of discrete events, in a same way that we model continuous processes such as changes in the real-pressure or the real-temperature by using discrete actions and ordinary variables. So, for example, if there is no system change between time x and time $x + 10$, we can pretend that time has progressed, 10 time-units in a single event.

3.5 Timed CSP

Timed CSP [96, 51] has been introduced by Reed and Roscoe [101] as a real-time extension of the CSP. In the initial version, the only change was adding *WAIT* t for any time t, to the primitives of CSP language.

Then, Schneider [105] and Davies [50] developed a proof systems for Timed CSP and added some new features to it such as time-outs and interrupts. Besides, Jackson [77]

has developed an approach for model checking timed CSP, by restricting the language to a finite-state version, and providing a temporal logic for it.

Refinement and parallel composition features are available in timed CSP and the FDR tool can be used for model checking some versions of finite-state timed CSP [97].

3.6 Timed Communicating Object-Z (TCOZ)

TCOZ [88, 4] is an integral formalism for complex systems. It has been developed by integrating the *Z* language [57] and *Communicating Sequential Processes (CSP)* [73]. Object-Z is an extension of Z language in order to facilitate formal specification, in an object oriented style. The Z notation [113] is based upon set theory and first-order predicate calculus. As mentioned in Section 2.6, in CSP a process is described by its possible interactions with its environment [98]. In TCOZ the Timed CSP has been used which is an extension of CSP process algebra notation for real-time systems [88].

Each of these modelling approaches has some advantages and disadvantages. The idea behind this combination, is that these two methods will complete each other's incompleteness. Object-Z has strong data and state modelling facilities, which have been gained by extending Z by the object oriented structuring techniques [56], but it has a single threaded semantic where operations are atomic. Whereas, timed CSP has a strong process control modelling capability, and its multi threads and synchronization primitives have been extended by timing primitives [56], but in compare to Z, it has not been sufficiently suited to modelling complex data structures, required for representing the states of a complex system.

A model in TCOZ has the same structure as an Object-Z model, which consists of a sequence of types and constants definitions in Z. But each operation should be defined in terms of a CSP process, which describes how that operation changes the state of that system.

Same as CSP (Section 2.6.1), each process in TCOZ can engage in a set of events. The CSP view of an operation in TCOZ, describes all the sequences of events, which change a system state. As a result, an update in a system state can have timing primitives.

Since TCOZ mostly preserves the syntax and semantics of each notation, it benefits from their methods and the tool supports, such as refinement and verification techniques.

3.7 Circus

Circus [114, 94] is a unified programming language containing both Z (model-based) and CSP (behavioural) constructs, specification statements and guarded commands.

Similar to TCOZ explained in Section 3.6, the idea is to combine two main approaches of applying formal techniques for precise and correct software development, in order to benefit from their advantages in a unified framework.

In Circus, concurrent programs can be modelled in terms of communicating abstract data types, by having all the existing combinations of Z with a process algebra. Besides, refinement feature is included, based on weakest preconditions and CSP.

In order to specify the timing aspects of real-time systems, Circus has been extended to includes the operators of Timed CSP [96, 51]. *Timed Circus* [94] only inherits the CSP part of Circus. Its syntax is very similar to timed CSP, but its semantics is based on a complete lattice in the implication ordering, which provides the required means to deal with temporal behaviours with multiple time scales.

3.8 Continuous Action Systems

Continuous action systems [26, 27] extend the action system approach to formalize hybrid systems. A hybrid system uses discrete control, over continuously evolving processes, whereas action systems use a discrete control upon a discrete state space.

This approach supports modelling of the real-time behaviours of a system, by ranging the state variables over time, based on some functions. As a result, a variable is not just a representation of the current state, but it captures the whole history of the values, it ever had, as well as the default values it will receive in the future. Consequently, updates are restricted to just changing the future behaviour of a variable.

In a continuous action system, reasoning about properties is based on refinement calculus, and the stepwise development is supported.

3.9 Real-time VDM

Vienna Development Method (VDM-SL) [80] is a formalism for specification of computer systems, where mathematical notations is used to precisely describe the desirable functionalities of a system. A system is modelled in terms of a state with a collection of operations, described as pre and post- conditions. Refinement is supported in VDM and the consistency of different abstract levels can be proved by discharging a number of proof obligations provided by the method.

VDM-SL is a flat language which is not sufficient for real-size systems specifications, whereas VDM++ [58, 86] extends it to support object oriented designs. In VDM++, a system is modelled as a collection of classes, where each class may contain values,

types, instance variables, functions and operations. In VDM++ concurrent processes are modelled by using threads, and their real-time behaviours can be analysed dynamically.

There are some tool supports for VDM++ such as VDMTools and Overture. These tools are model checkers which evaluate invariants and pre/post-conditions, and supporting static analysis of models.

Some scheduling algorithms are supported by VDMTools and Overture. For example, it is possible to specify an execution period for a statement or define a periodic thread. In order to evaluate the real-time properties of a model, information of its real-time behaviour is gathered during its execution. The objective of performing timing analysis in these tools, is to specify those parts of a system, which their performances have the potential to cause a deadline violation. This is done by having *Cycle/Duration* tags, which can be used to specify the required duration in order to execute a segment of a model. A *Cycle* tag specifies the required CPU's cycles, in order to accomplish the corresponding segment. So it is a relative constraint based on the strength of the CPU. But the *Duration* tag specify a specific duration of time for a segment to be executed.

There is no support for the refinement feature in VDM++ real-time modelling. Existing works on modelling real-time systems in VDM++, mostly focus on validating the potential candidate of a system's architect, in the early stages of system development process [108].

3.10 VDM++ Combined by Co-simulation

VDM++ Combined by Co-simulation is introduced by Verhoef [111] which supports modelling of both discrete and continuous behaviours. So, it is possible to model a control system discretely, and have a continuous representation of its environment. Also, VDM++ is extended to support asynchronous actions and parallel processes too.

By this approach, each process unite in the controller, is modelled by a process in the VDM++ and its timing properties is specified in a discrete timing system. A controller model is connected to a model of the target environment which has been modelled in a Co-simulator by a set of continues timing properties and behaviours. As a result the environment changes are continuous in a model, and the controller traces those changes by sampling the environment periodically and reacts, based on its internal discrete clock.

3.11 Modelling Timing in the B-method

B [106] is a step-wise formal method for specification and development of computer software systems. Functionality of a system is modelled in terms of a collection of

operations. There are a lot of similarities between B method and Event-B, both in modelling and proofing methods. Butler in [41] introduced an approach in order to model timing constraints in classical-B. In his work the only timing constraint which has been investigated is deadline. His approach to model time and deadline in B had an effect on our approach represented in this thesis.

Based on Butler's approach, a variable is required to represent the current value of time, and there is an event which represent the tick. This event increases the current value of time by a time-unit in each execution.

In order to enforce an event to occur before a specific time, the tick event is guarded according to the occurrence of the restricted event. Consequently, if that event has not occurred yet and by occurrence of the tick event, the current time value will exceed the specified upper bound, the tick event will be disabled.

By this approach, a system's global clock and its deadlines can be encoded in a B method, but their refinement has not been investigated.

3.12 Real-time Event-B

There are some existing works on modelling timing properties in Event-B. Cansell et al. [44] modelled a real-time leader election protocol in Event-B. In that study **Time Constraint Pattern** (TCP) was introduced in order to model time and express timing constraints in Event-B. In TCP, the time progress is an event and no modification has been done on the underlying language of Event-B. According to this approach, for each event which has to happen in a specific time, a set will keep track of its activation times. The event which increase time is guarded by those sets to prevent it from happening, if its occurrence will makes the current time value, greater than the minimum of those sets' union.

EVENT *SendM* $\widehat{=}$
any m **WHERE**
$m \in MSG$
$m \notin dom(timesent)$
THEN
$timesent := timesent \cup \{m \mapsto now\}$
$deadline := deadline \cup \{m \mapsto now + limit\}$
$q\_rcv := q\_rcv \cup \{m\}$
END

(a) Event *SendM* represents the send message process

EVENT *Receive* $\widehat{=}$
any m **WHERE**
$m \in q\_rcv$
$deadline(m) \geq now$
THEN
$deadline := deadline \setminus \{m \mapsto deadline(m)\}$
$ontime := ontime \cup \{m\}$
$q\_rcv := q\_rcv \setminus \{m\}$
END

(b) Event *Receive* represents the on-time receiving of the message

EVENT *ReceiveLate* $\widehat{=}$
any m **WHERE**
$m \in q\_rcv$
$deadline(m) < now$
THEN
$deadline := deadline \setminus \{m \mapsto deadline(m)\}$
$late := late \cup \{m\}$
$q\_rcv := q\_rcv \setminus \{m\}$
END

(c) Event *ReceiveLate* represents the late receiving of the message

Figure 3.3: Events *SendM*, *Receive* and *ReceiveLate* according to Bryans approach.

When time gets to the activation time of an event, the event will become eligible to occur and by its occurrence the activation time will be removed from the corresponding set. In this way, when all the events whose activation time is equal to the current time value have happened, the time progressing event becomes eligible to happen again. An activation time is added by the event which triggers the corresponding timing constraint.

By using TCP and modifying it according to the needs of a specific communication protocol, Bryans et al.[37] modelled a message passing protocol and its timing properties, which has upper bound, lower bound, and recovery scenarios for the messages which have not been received by their expiry time. The base of the approach is similar to the TCP, the only difference is that the activation time mechanism has been changed to the upper and lower bounds, where events are not forced to happen in a specific time, and if they do not occur by their upper bound, they will not be eligible to occur any more, and their alternative event which is constraint by a minimum delay (lower bound) will become eligible to happen. Besides, the time forwarding event is not guarded anymore and timing constraints encoded as guards on their corresponding constrained events.

In their case study, the value of the deadline set for each message, acts as an expiry limitation for the event which represent the normal scenario, and acts as a delay for the

recovery scenario. As a result, by this approach it is possible to model upper bounds (in our approach is called expiry) or delays for events' occurrences. As shown in Figure 3.3, event *SendM* adds the time of each message to the *deadline* set, then, if event *Receive* for that message, does not happen before the added time, event *Receive* will be disabled and event *ReceiveLate* will be eligible to occur.

In this chapter some of the existing works on modelling real-time systems and reasoning about them, have been introduced briefly. In the following chapters, our approach of modelling timing properties in Event-B, and the developed case studies based on them will be discussed in details.

Chapter 4

Modelling Timing Properties In Event-B

In this Chapter, first three groups of timing properties which are the main focus of this research will be introduced. Then our approach to formulate their Event-B representation, will be discussed in details. The formulation approach consists of some constructs, to express the timing properties, and their translation to invariants, guards and actions. In the end, since refinement is one of the most important features of Event-B, some patterns to refine abstract timing properties to concrete ones based on the control flow refinement, will be introduced.

4.1 Time Properties Categories

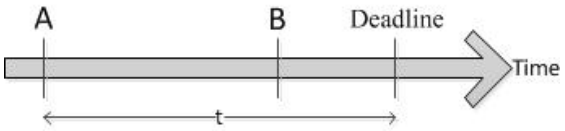
In order to explicitly represent timing properties we extend the Event-B syntax with constructs for deadlines, delays and expiries. A typical pattern is a trigger followed by its possible responses, thus each of our timing constructs specifies a constraint between a trigger event A, and either a response event B, or a set of response events $B_1..B_n$. The syntax for each of these properties is as follows:

$$\textbf{Deadline}(A, B_1 \vee .. \vee B_n, t)\,, \tag{4.1a}$$

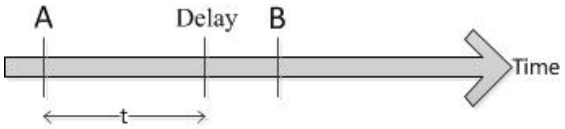
$$\textbf{Delay}(A, B, t)\,, \tag{4.1b}$$

$$\textbf{Expiry}(A, B, t)\,. \tag{4.1c}$$

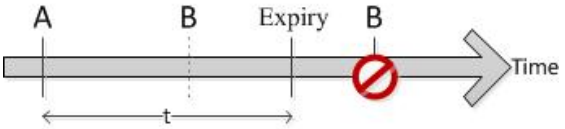
Deadline$(A, B_1 \vee .. \vee B_n, t)$ (4.1a) means that one and only one of the response events $(B_1..B_n)$ must occur within time t of trigger event (A) occurrence (Figure 4.1(a)). We use a disjunction symbol between the possible responses, to indicate the alternative nature of the property, where any of the responses's occurrence will satisfy the property.



(a) Timing diagram of a deadline property.



(b) Timing diagram of a delay property.



(c) Timing diagram of an expiry property. Event B may only occur before the expiry duration (t). That is why it is represented by a dash line.

Figure 4.1: In these diagrams, t is the timing property's duration, A is the trigger event and B is its response event, and the horizontal axis is the time line.

In the case of delay (4.1b), the response event can only happen if the delay period has passed following an occurrence of the trigger event (Figure 4.1(b)). Finally the expiry (4.1c) means that the response event cannot happen if the expiry period has been passed following an occurrence of the trigger event (Figure 4.1(c)).

Our experiences in modelling of real-time case-studies, show that the focus, in the abstract levels, is deadline properties, and sometimes some expiry properties are required to prove the consistency of the refinements of those deadlines. Whereas, delay properties usually appear in the more concrete levels to present the detailed properties of a system's control flow. For example in the automatic gear controller case study, explained in Chapter 7, in the most abstract level, the maximum duration required for the system to respond a gear change request has been modelled, and the following refinements aim to prove that the deadlines of concrete sub-steps required in a gear changing process are consistent with the overall deadline. In the end, when all the concrete steps have been added to the model, some delays have been introduced to precisely express the order of alternative steps.

The syntax presented in this section to express the timing properties of a system, help to systematise the process of specifying discrete timing properties in Event-B models, and hide the complexity of encoding timing properties in an Event-B model from the modeller. As a result, a timed-Event-B machine from modeller point of view, will be a non-timed-Event-B machine plus a list of its timing properties, declared by the introduced constructs.

In the following sections, the semantics of these timing extensions of Event-B will be discussed.

4.2 Semantics of Timing Properties In Event-B

We give a semantics to our timing constructs by translating them into Event-B variables, invariants, guards and actions. In particular, these timed-Event-B elements constrain the order between trigger event, response events and the time-progressing event ($Tick\_Tock$).

In each case we assume there is already a partial order between the trigger event and the corresponding response events. The assumption is that the response events are only enabled if the trigger event has already happened. This ordering assumption for a sequential control flow, is encoded by using boolean flags for unparametrized events. As shown in Figure 4.2(a), event A sets the boolean variable fA as one of its actions. So when variable fA has the value of $TRUE$, indicates that event A has already happened. Also, one of the response events' guards, checks the occurrence of trigger event A, by evaluating its occurrence flag.

It has not been assumed that the trigger and response events will occur only once. Typically the trigger and the response events are part of an iterative behaviour. When the steps' sequence of an iterative behaviour, have been modelled by boolean flags, those flags have to be reset at the end of each iteration, to provide the required initial state, for the following iteration. Imagine a model consists of a request event and a response event, where an occurrence of the request event has to be followed by an occurrence of the response event and this process may iterate indefinitely. To model this behaviour by boolean flags, we add another event which happens after the response event and reset the occurrence flags of the request and response event. The resetting event provides the initial states of the next iteration.

In Section 2.8.4.1 we talked about the consistency of Event-B machines, and the corresponding PO. Based on that, in the following sections we will prove the consistency of each timing property's semantics. For each timing property, how the corresponding POs of its invariants will be discharged will be explained in details.

4.2.1 Delay Semantics

In this section we explain how delay is encoded in an Event-B model. As mentioned before, in order to have discrete time in Event-B a natural number variable is declared to represent the current value of time, and an event is added to model the progress of time.

In order to explain the semantics of delay in Event-B, we assume a generic trigger event A and a generic response event B. In the Patterns of this section, $G_X(c, v)$ represents the guards of event X, and Act_X represents its actions, where c denotes the constants and v the variables of the corresponding model.

A delay property is structured as follows:

$$Delay(A, B, t)\,. \tag{4.2}$$

There are three parts to the Event-B semantics of a delay property. First the occurrence time of the trigger event is recorded in a variable (tA). Second, in the response event (B), a guard is needed which disables the response event if the stated delay duration has not been passed, from the occurrence of the trigger event, and an action is added to record its occurrence in a variable (tB). The occurrence time of the response event is required for the delay invariants which is the last part of a delay semantics.

TIMING
$Delay(A, B, t)$
EVENT $A \widehat{=}$
WHERE
$fA = FALSE$
$G_A(c, v)$
THEN
$fA := TRUE$
Act_A
END
EVENT $B \widehat{=}$
WHERE
$fA = TRUE$
$fB = FALSE$
$G_B(c, v)$
THEN
$fB := TRUE$
Act_B
END

(a) Event A and B plus a delay

INVARIANTS
(inv_1) $\neg fA \Rightarrow \neg fB$
(inv_2) $fB \Rightarrow tB \geq tA + t$
EVENT $A \widehat{=}$
WHERE
$fA = FALSE$
$G_A(c, v)$
THEN
$fA := TRUE$
$tA := time$
Act_A
END
EVENT $B \widehat{=}$
WHERE
$fA = TRUE$
$fB = FALSE$
$time \geq tA + t$
$G_B(c, v)$
THEN
$fB := TRUE$
$tB := time$
Act_B
END

EVENT $Tick\_Tock \widehat{=}$
any
$tick$
WHERE
$tick > 0$
THEN
$time := time + tick$
END

(b) Encoded delay for events A and B.

Figure 4.2: Semantics of a delay property in Event-B.

In Figure 4.2(a) shows the generic trigger-response pattern plus a delay property, and in Figure 4.2(b) shows how the delay is represented in terms of standard Event-B elements. As shown in Figure 4.2(b), delay semantics included two invariants, inv_1 specifies that the response event always happens after the trigger event, and inv_2 express the property that if response B has happened, its occurrence time must exceed the occurrence time of trigger event A by at least t. inv_1 is required to discharge the corresponding POs of inv_2. Besides, the $Tick\_Tock$ event is a part of the semantics of timing properties which models the progress of time based on the $tick$ duration.

In the following, the consistency of the delay semantics will be proved. In the proofs, we will just presents the selected hypothesis, required to discharge the PO. For a machine The consistency of invariant inv_1 in Figure 4.2(b), can be proved as follows:

For event INITIALISATION: (From left to right)

fA' =FALSE fB' =FALSE ⊢ fA' =FALSE ⇒fB' =FALSE	$fA' = \text{FALSE}$ $fB' = \text{FALSE}$ ⊢ FALSE=FALSE ⇒ FALSE=FALSE	$fA' = \text{FALSE}$ $fB' = \text{FALSE}$ ⊢ ⊤ ⇒ ⊤

For event A: (From left to right)

fA =FALSE fA' =TRUE ⊢ fA' =FALSE ⇒fB =FALSE	fA =FALSE fA' =TRUE ⊢ TRUE=FALSE ⇒fB =FALSE	fA =FALSE fA' =TRUE ⊢ ⊥ ⇒ fB =FALSE ≡ ⊤

For event B: (From left to right)

fA =TRUE fB =FALSE fB' =TRUE ⊢ fA =FALSE ⇒fB' =FALSE	(Deduction) fA =TRUE fB =FALSE fB' =TRUE fA =FALSE ⊢ fB' =FALSE	Contradiction in hypothesis

Since event *Tick_Tock* does not change any of the variables involve in the invariant, there is no need to prove the invariant preservation.

For invariant $fB \Rightarrow tB \geq tA + t$ (inv_2) the consistency proof will be as follows:

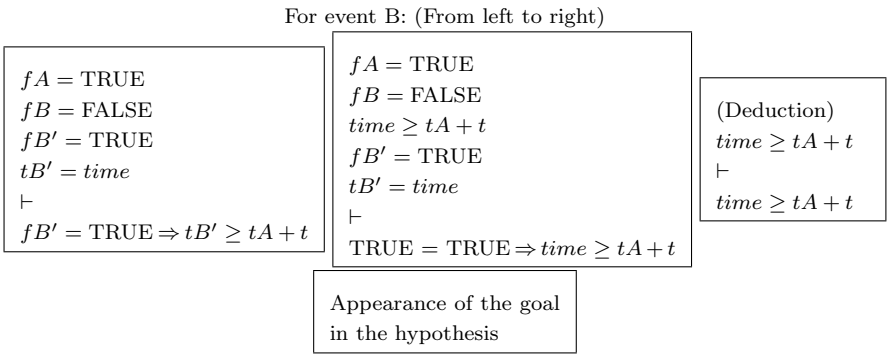
For event INITIALISATION:

$tA' = 0$ $tB' = 0$ $fB' = \text{FALSE}$ $t > 0$ ⊢ $fB' = \text{TRUE} \Rightarrow tB' \geq tA' + t$	$tA' = 0$ $tB' = 0$ $fB' = \text{FALSE}$ $t > 0$ ⊢ FALSE=TRUE ⇒ $0 \geq 0 + t$	$tA' = 0$ $tB' = 0$ $fB' = \text{FALSE}$ $t > 0$ ⊢ ⊥ ⇒ ⊥ ≡ ⊤

For event A: (From left to right)

$fA = \text{FALSE}$ $fA = \text{FALSE} \Rightarrow fB = \text{FALSE}$ $fA' = \text{TRUE}$ $tA' = time$ ⊢ $fB = \text{TRUE} \Rightarrow tB \geq tA' + t$	(Deduction) $fA = \text{FALSE}$ $fA = \text{FALSE} \Rightarrow fB = \text{FALSE}$ $fA' = \text{TRUE}$ $tA' = time$ $fB = \text{TRUE}$ ⊢ $tB \geq time + t$	$fA = \text{FALSE}$ $\top \Rightarrow fB = \text{FALSE}$ $fB = \text{TRUE}$ ⊢ $tB \geq time + t$	$fA = \text{FALSE}$ $fB = \text{FALSE}$ $fB = \text{TRUE}$ ⊢ $tB \geq time + t$

Contradiction in hypothesis

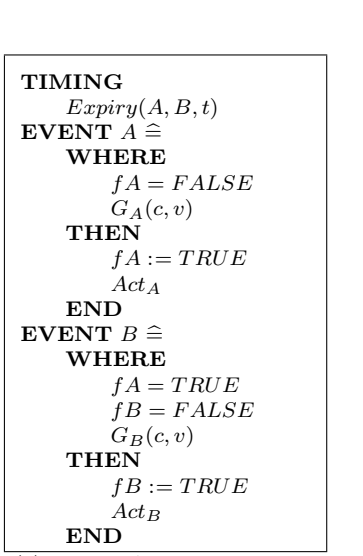


Again, since the *Tick_Tock* event does not change any of the variables involve in the invariant, the will be preserved by its occurrence, and no proof is required.

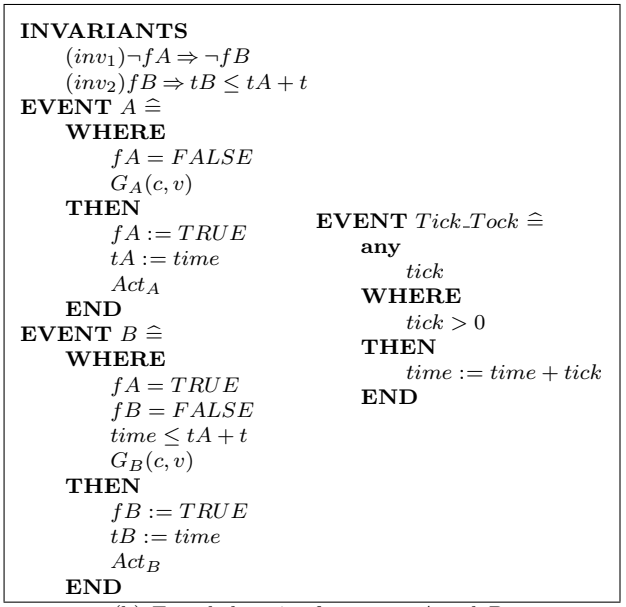
So far the semantics of delay has been presented and its consistency has been proved. In the next section the semantics of expiry will be introduced.

4.2.2 Expiry Semantics

Expiry is given an Event-B semantics similar to delay, guarding the response events according to the recorded occurrence time of the trigger event and the specified expiry period.



(a) Events A and B plus an expiry



(b) Encoded expiry for events A and B.

Figure 4.3: Semantics of an expiry property in Event-B.

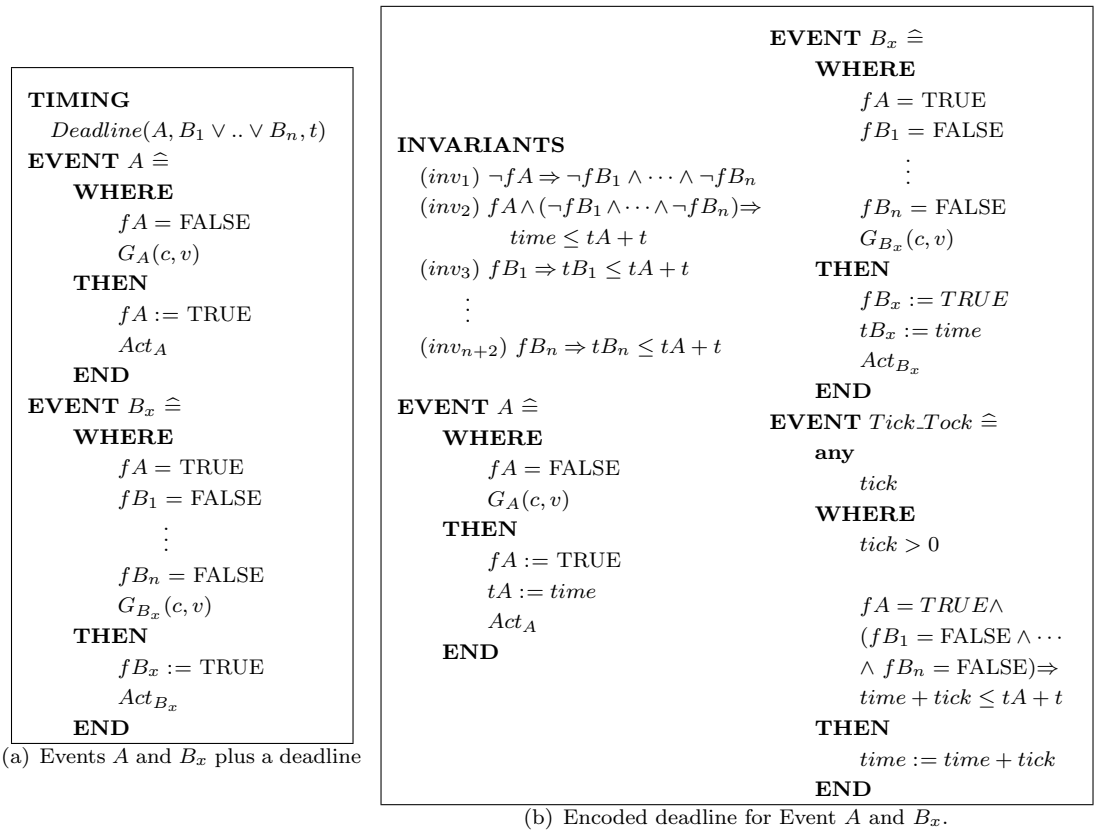
In order to explain how expiry is represented in Event-B, we assume a generic trigger event A and its generic response event B, with an expiry as shown in Figure 4.3(a).

As shown in Figure 4.3(b), in order to have an expiry on a trigger-response pattern, an action is needed to record the occurrence time, in the trigger event (event A), and a guard on the response event, to prevent it from happening, if the expiry period has been passed. Besides, the occurrence time of the response event is recorded which will be used in the expiry invariants.

Similar to the delay semantics, we have two invariants as a part of expiry semantics, one express the order between trigger and response event, and the other express that if the response B has happened, its occurrence time should not exceed the occurrence time of A by at most t. These two invariants can be proved in a same way the delay's invariants have been proved.

4.2.3 Deadline Semantics

As explained in the introduction section, in time-critical systems, there are assumptions about the maximum duration required to establish different processes. These assumptions can be modelled by deadlines in our approach.



(a) Events A and B_x plus a deadline

(b) Encoded deadline for Event A and B_x.

Figure 4.4: Semantics of a deadline property in Event-B.

Based on the semantics of delay and expiry, it is the response event, which will be guarded based on time. However, according to the deadline semantics, the $Tick\_Tock$ event is

guarded instead. If the trigger event has happened, we want to force one and only one of the response events to occur, before passing the deadline. Guarding the $Tick\_Tock$ event is a way of enforcing one of the response events to occur, before passing the deadline. So, by guarding the $Tick\_Tock$ event, the upper bounds assumption can be enforced to a model.

Assume a deadline property, structured as follows:

$$Deadline(A, B_1 \vee .. \vee B_n, t)\,. \tag{4.3}$$

The guard on the $Tick\_Tock$ event to enforce deadline (4.3), prevents reaching the deadline ($time + tick \leq tA + t$), if trigger event A has occurred but a response event B_x has yet to occur (Figure 4.4(b)).

Similar to delay and expiry, deadline semantics contains several invariants. As shown in Figure 4.4(b), inv_1 expresses the order between trigger event and its alternative responses. Based on this invariant, none of the responses can happen unless the trigger event has already happened. inv_2 specifies the deadline property that if the trigger event A has happened, but none of its responses has happened yet, then time should not exceed the occurrence time of A by at most t.

inv_2 talks about the value of the current time based on the state of trigger and response events. On the other hand, $inv_3 \cdots inv_{n+2}$ express the deadline property for the occurrence time of each response event. Based on these invariants, if any of the response events has happened, its occurrence time should not exceed the occurrence time of A by at most t.

In the following the proofs of the deadline semantics consistency will be presented in details. For invariant inv_1 based on Figure 4.4(b), the proof is as follows:

For event INITIALISATION: (From left to right)

fA' =FALSE fB'_1 =FALSE ⋮ fB'_n =FALSE ⊢ fA' = FALSE ⇒fB'_1 =FALSE $\wedge \cdots \wedge fB'_n$ = FALSE	fA' =FALSE fB'_1 =FALSE ⋮ fB'_n =FALSE ⊢ FALSE = FALSE ⇒ FALSE = FALSE $\wedge \cdots \wedge$ FALSE = FALSE $\equiv \top$

For event A: (From left to right)

fA =FALSE fA' =TRUE ⊢ fA' = FALSE ⇒fB_1 =FALSE $\wedge \cdots \wedge fB_n$ = FALSE	fA =FALSE fA' =TRUE ⊢ TRUE=FALSE ⇒fB_1 =FALSE $\wedge \cdots \wedge fB_n$ = FALSE	fA =FALSE fA' =TRUE ⊢ $\bot$⇒fB_1 =FALSE $\wedge \cdots \wedge fB_n$ = FALSE $\equiv \top$

For event B_x: (From left to right)

fA =TRUE
fB_x =FALSE
fB'_x =TRUE
⊢
fA = FALSE ⇒fB'_x =FALSE ∧ none-primed flags of all the other responses are FALSE

(Deduction)
fA =TRUE
fB_x =FALSE
fB'_x =TRUE
fA = FALSE
⊢
fB'_x =FALSE ∧ none-primed flags of all the other responses are FALSE

Contradiction in hypothesis

Since the *Tick_Tock* event does not modify any state variables involved in this invariant, the invariant will be preserved.

For invariant inv_2 which expresses a property of the current time based on the deadline, the consistency proof is as follows:

For event INITIALISATION: (From left to right)

fA' =FALSE
fB'_1 =FALSE
⋮
fB'_n =FALSE
tA' =0
$time'$ =0
$t > 0$
⊢
fA' = TRUE ∧fB'_1 =FALSE ∧ ⋯ ∧ fB'_n = FALSE ⇒ $time' \leq tA' + t$

fA' =FALSE
fB'_1 =FALSE
⋮
fB'_n =FALSE
tA' =0
$time'$ =0
$t > 0$
⊢
FALSE = TRUE ∧ FALSE = FALSE ∧ ⋯ ∧ FALSE = FALSE ⇒$0 \leq 0 + t$

fA' =FALSE
fB'_1 =FALSE
⋮
fB'_n =FALSE
tA' =0
$time'$ =0
$t > 0$
⊢
$\bot \Rightarrow \top$

For event A: (From left to right)

fA = FALSE
fA' = TRUE
$tA' = time$
$t > 0$
⊢
fA' = TRUE ∧ fB_1 = FALSE ∧ ⋯ ∧ fB_n = FALSE ⇒ $time \leq tA' + t$

(Deduction)
fA = FALSE
fA' = TRUE
$tA' = time$
$t > 0$
fB_1 = FALSE ∧ ⋯ ∧ fB_n = FALSE
⊢
$time \leq time + t \equiv \top$

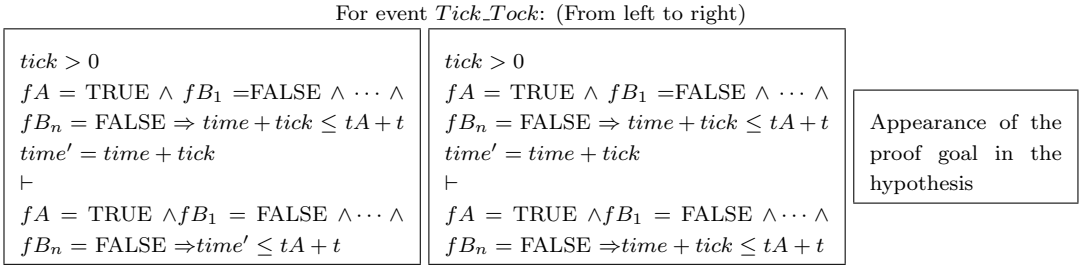
For event B_x: (From left to right)

fA = TRUE
fB_1 = FALSE
⋮
fB_n = FALSE
fB'_x = TRUE
$t > 0$
⊢
fA = TRUE ∧ fB'_x = FALSE∧ none-primed flags of all the other responses are FALSE ⇒ $time \leq tA + t$

fA = TRUE
fB_1 = FALSE
⋮
fB_n = FALSE
fB'_x = TRUE
$t > 0$
⊢
fA = TRUE ∧ $TRUE$ = FALSE∧ none-primed flags of all the other responses are FALSE ⇒ $time \leq tA + t$

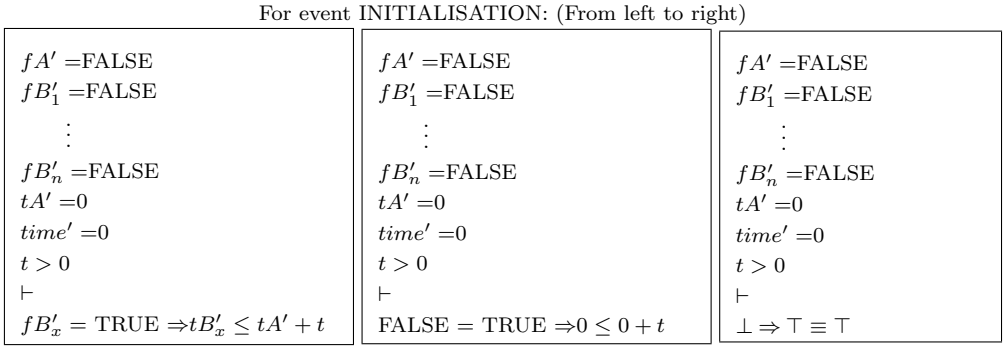
fA = TRUE
fB_1 = FALSE
⋮
fB_n = FALSE
fB'_x = TRUE
$t > 0$
⊢
$\bot \Rightarrow time \leq tA + t \equiv \top$

For event *Tick_Tock*: (From left to right)

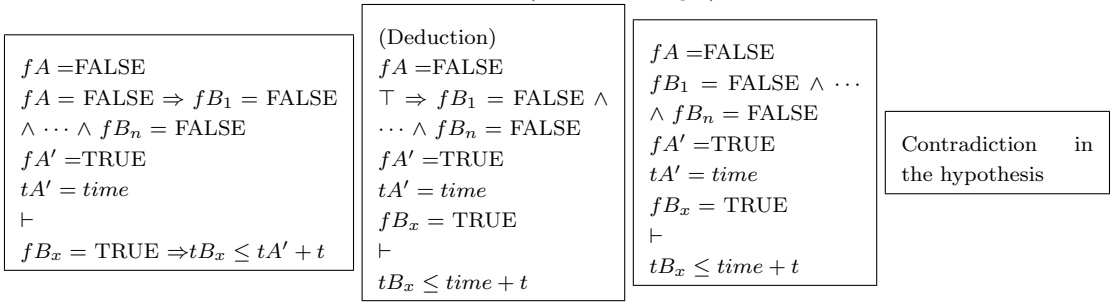


For invariants inv_3 to inv_{n+2}, the proof will be similar. So we will just go through the proof of the invariant which expresses a property of response event B_x occurrence time ($x \in 3..n+2$).

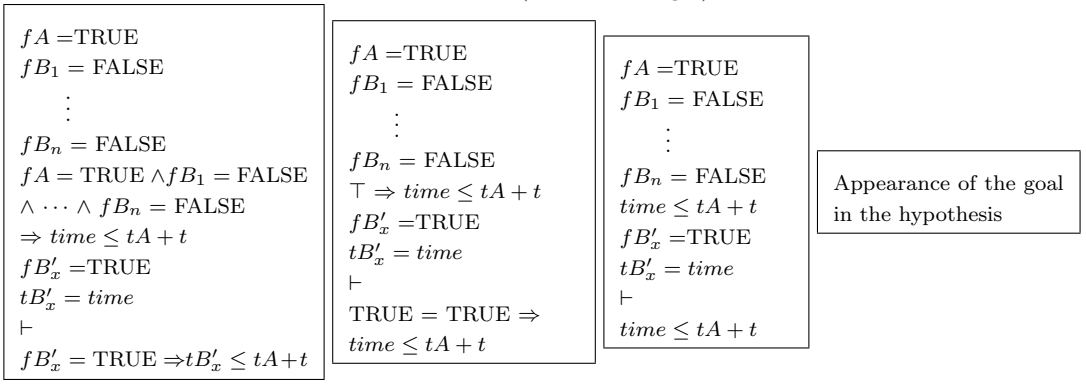
For event INITIALISATION: (From left to right)



For event A: (From left to right)



For event B_x: (From left to right)



For any response event B_y ($y \neq x$), since the event will not change any of the variables included in the invariants, it will be preserved by the response event. The same is true

about the *Tick_Tock* event. Based the proofs presented in this section, we have shown that the deadline semantics is consistent.

Multiple deadline properties may be added to a model. In this case, a deadline guard similar to what has been shown in Figure 4.4(b), should be added to the *Tick_Tock* event, for each deadline property.

So far the syntax and the semantics of delay, expiry, and deadline have been presented. As mentioned before, one of the most important features of Event-B is refinement, and it has been our intention to provide a semantics for timing properties which supports this feature. In the following we will present some refinement patterns, in which the timing properties of an abstract behaviour, have been replaced by the timing properties of its concrete behaviour, while the refinement consistency has been preserved.

4.3 Some Patterns to Refine Deadline, Delay and Expiry

In this section, some patterns to refine an abstract deadline or an abstract expiry, to more detailed timing properties will be explained. It should be mentioned that these are not modelling patterns, rather they are refinement patterns; the aim of our patterns is to explain how timing properties can be refined based on some specific control flow refinement patterns.

Each refinement pattern will be explained by applying it to a generic control flow refinement pattern. Besides, the gluing invariants, required to discharge the refinement consistency POs will be discussed for each refinement pattern. The assumption is that the control flow refinement without the timing properties is consistent. For example, if an abstract response event has been refined by two sequential concrete sub-responses, we assume that the refinement has been consistent, and its correctness has been already proved. So we will be focusing on how the timing properties can be refined accordingly, and what gluing invariants are required to prove the preservation of the abstract timing properties by the concrete ones.

In the rest of this report timed-Event-B models will be shown from a modeller point of view. So, each timed-machine will be a list of its timing properties specified by the introduced constructs in Section 4.2, plus the non-timed Event-B machine. In each refinement pattern, for the constants c and set of variables v of a machine, $G_X(c, v)$ presents the guards of event X in that machine and Act_X presents the actions of that event.

Besides, in this section, refinement diagrams (Section 2.10) have been used to present the refinement relations of events, in different levels of abstraction.

4.3.1 Refining a Deadline to Sequential Sub-Deadlines

Consider an abstract model of a system where there is a deadline between event A and event B. As shown in Figure 4.5, event B can only occur if event A has already happened. The deadline property of this level of abstraction, is shown in Figure 4.6(a).

As shown in Figure 4.5, event B has been broken to two sequential steps, in the refinement. By breaking event B to B_1 followed by B_2, its related deadline needs to be broken too. The other important issue in this pattern is that, the abstract event has been refined by the second step, because the accomplishment of the second step is equivalent to accomplishment of the abstract event (B). So the first step should refine *skip*.

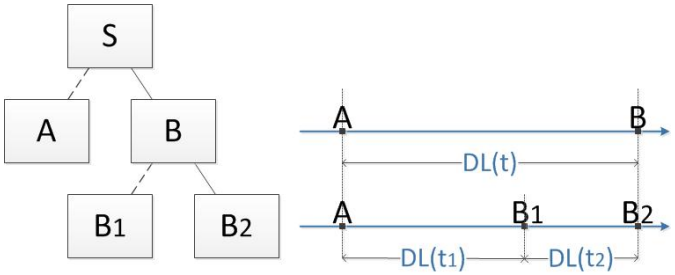


Figure 4.5: Refining an abstract deadline to two sub-deadlines is presented by the refinement diagram on the left. $DL(x)$ presents a deadline property with a period of x in the timing diagrams.

Now, in order to respond to the trigger event, two steps have to be accomplished, where each of them has its own deadline. In the concrete level, the trigger event of the deadline property for event B_1 is event A and the trigger event for the deadline of event B_2 is event B_1. Hence, the abstract deadline should be broken into two new deadlines, in a way that their combination, based on the concrete order, does not violate the abstract deadline ($t_1 + t_2 \leq t$).

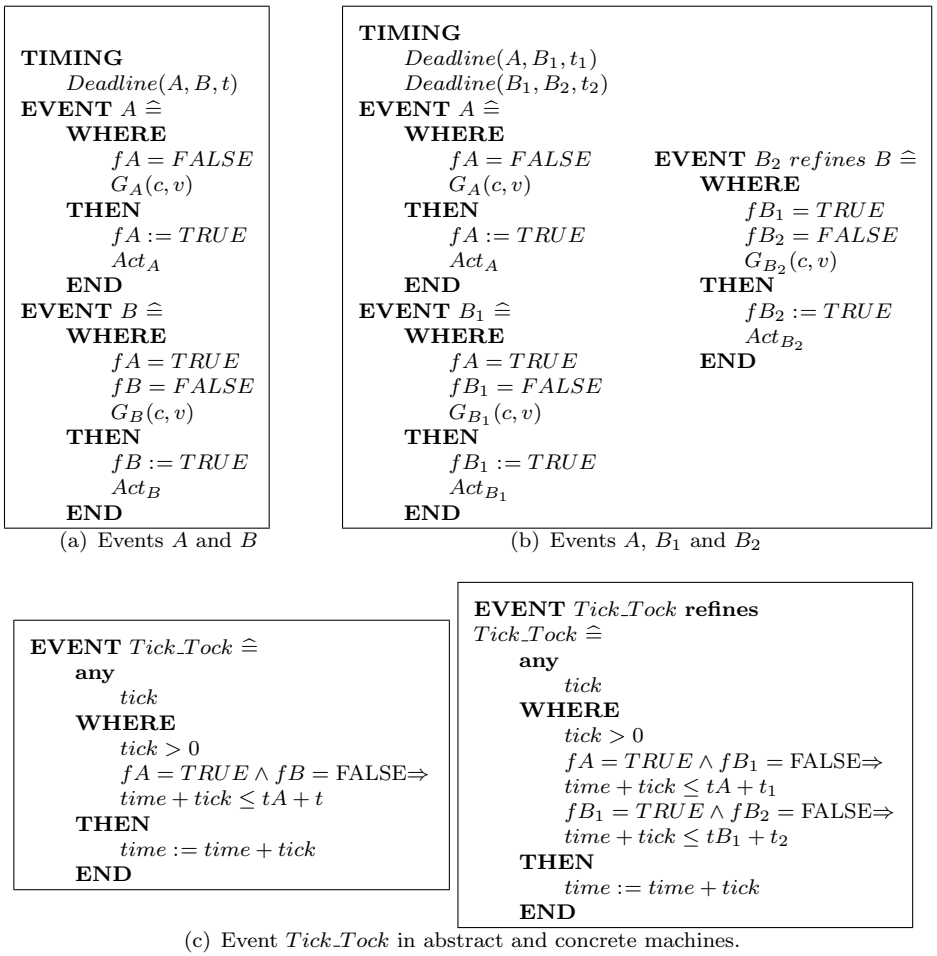
We need to prove that the concrete machine refines the behaviour of its abstract one. For the refinement pattern, presented in Figure 4.5, it is necessary to prove the abstract deadline holds in the concrete machine.

As shown in Figure 4.6, in the concrete machine, the abstract deadline between event A and event B is refined by the following deadlines:

$$Deadline\ (A,\ B_1,\ t_1)\,, \tag{4.4a}$$

$$Deadline\ (B_1,\ B_2,\ t_2)\,. \tag{4.4b}$$

Based on deadline (4.4b) if event B_1 has happened and event B_2 has not happened yet, then the current value of time should be less than or equal to the occurrence time of



(a) Events A and B

(b) Events A, B_1 and B_2

(c) Event $Tick\_Tock$ in abstract and concrete machines.

Figure 4.6: Events A and B plus their deadline property in the abstract Machine in 4.6(a), followed by event A , events B_1 and B_2 in the concrete machine plus their concrete timing properties in 4.6(b). As mentioned before, the $Tick\_Tock$ event is part of the semantics, but we have presented it to clarify the refinement.

event B_1 plus the deadline period (t_1). By having this timing property the relation between the occurrence time of event B_2 and event B_1 has been specified. But we are interested on the relation of the occurrence time of event B_2 and event A. So it is enough to specify the relation of the occurrence time of event B_1 and event A.

The relation between the concrete states and the abstract ones is expressed by *gluing invariants* [13] in Event-B, in order to verify a refinement. Two kinds of gluing invariants are needed, in order to prove that the concrete deadlines satisfy their abstract. The first type is needed to clarify the relation between the order in the abstract machine and the order in the concrete machine, which has not been modelled by explicit guards. For example based on the guards in the presented refinement pattern (Figure 4.6), B_2 can only happen after B_1 occurrence and B_1 can only happen after A occurrence, accordingly B_2 can only happen after A, but this property has not been mentioned explicitly in the guards of B_2. The other type of gluing invariants is needed to specify the relation

between the new deadlines in the concrete machine and the abstract deadline. For the refinement pattern presented in Figure 4.6, these invariants should be as follows:

- The relation between the abstract event and its refining event (B_2 and B are the boolean variables which act as the occurrence flags of events B_2 and B):

$$fB_2 = fB\,, \tag{4.5}$$

- The order between the concrete events (part of deadline semantics):

$$fA = FALSE \Rightarrow fB_1 = FALSE \tag{4.6}$$

- The relation between the abstract trigger event's occurrence time, and the occurrence times of the concrete trigger events (parts of deadline semantics):

$$fB_1 = TRUE \Rightarrow tB_1 \leq tA + t_1\,, \tag{4.7a}$$
$$fA = TRUE \wedge fB_1 = FALSE \Rightarrow time \leq tA + t_1\,. \tag{4.7b}$$

In the above invariants, tA is an integer variable which records the occurrence time of event A, and tB_1 does the same thing for event B_1. Invariant (4.5) specifies that the occurrence of event B_2 is equivalent to the occurrence of event B. This invariant is required for the control flow refinement, and it will be required in the untimed model too.

The relation of the occurrence time of event B_1 and event A has been specified by the gluing invariant (4.7a) based on deadline (4.4a). Based on invariant (4.7a) we know that if B_1 has occurred, then the duration between A, and B_1 does not exceed t_1.

Invariant (4.7b) which is equivalent to the required guard on the $Tick\_Tock$ event for deadline (4.4a) provides the required information to discharge the proof obligation of invariant (4.7a) for event B_1.

As mention in Section 4.2.3, both of invariants (4.7a) and (4.7b) are part of the deadline semantics, and their consistency has been demonstrated in that section. In this refinement the main challenge is to show that the deadline guard of the abstract model, will be satisfied by the deadline guards of the concrete machine. In Figure 4.7 the proof of this property will be presented to demonstrate how the introduced invariants facilitate the process.

It should be noted that the abstract deadline can be broken into more than two sub-deadlines either by successive refinement steps or by refining the abstract event with more than two sub-sequential events in one refinement step. For these refinement cases, it is possible to follow a similar approach.

In an Event-B model of this refinement, 14 POs have been generated for the abstract machine which all were proved automatically. For the concrete machine, 29 POs have been generated from which only one has been discharged interactively.

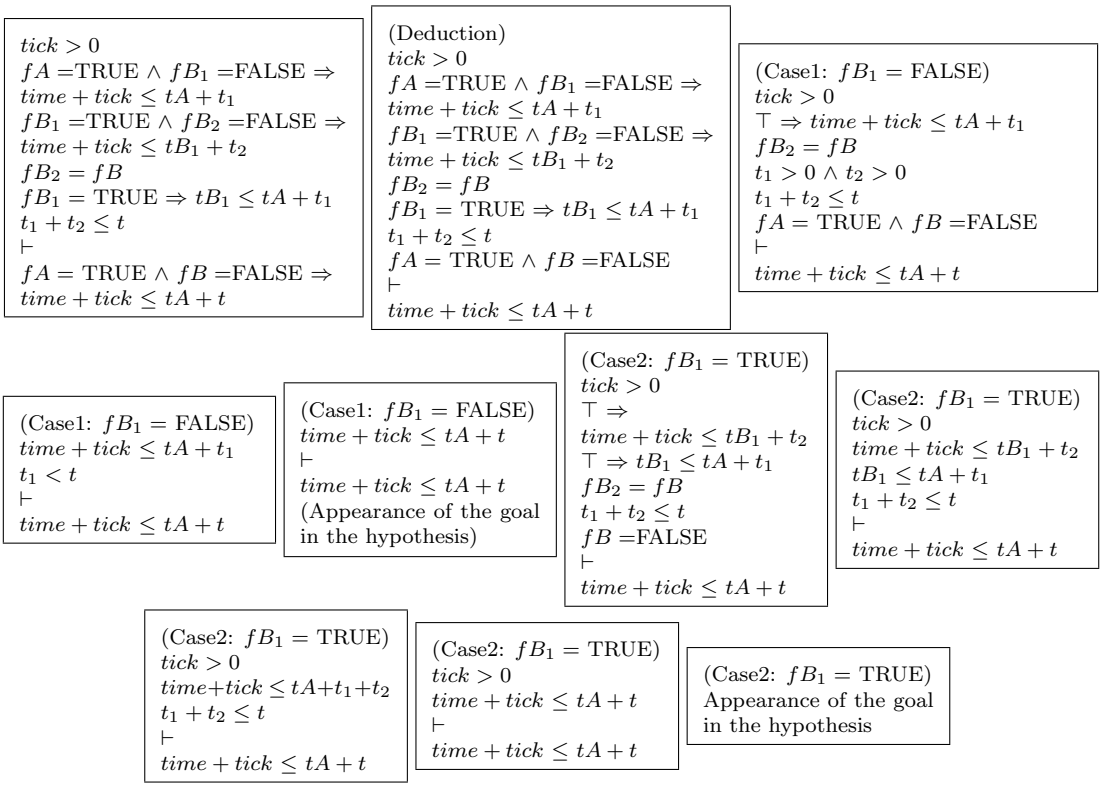


Figure 4.7: The proof of the property that the concrete deadlines' guards on the *Tick_Tock* event, preserve the abstract deadline's guard.

4.3.2 Refining an Expiry to a Sequence of an Expiry and a Deadline

Consider an abstract model of a system, where there is an expiry between a trigger event A, and its response event B. The expiry property of this level of abstraction, is shown in Figure 4.9. In the next refinement, event B has been broken into two sequential steps, as shown in Figure 4.8.

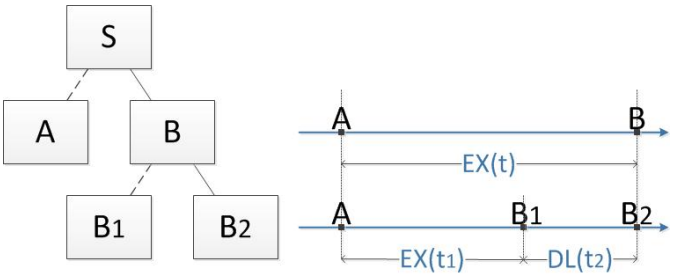
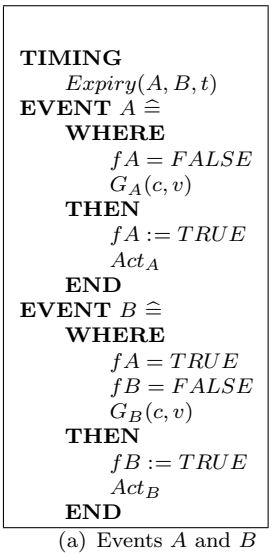


Figure 4.8: Refining an abstract expiry by a sequence of an expiry and a deadline is presented by a refinement diagram on the left. $EX(x)$ presents an expiry property with a period of x in the timing diagrams.

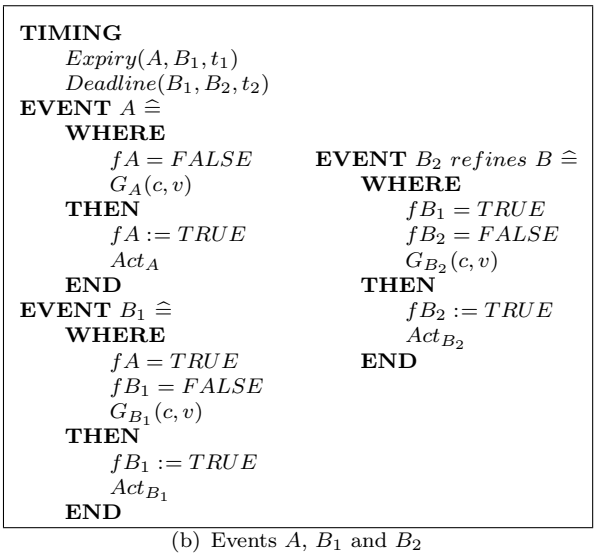
By breaking event B to B_1 followed by B_2, its related expiry needs to be satisfied by their timing properties. Since the concrete events are sequential, the accomplishment of

the last event (B_2), is equivalent to the occurrence of the abstract response event, and the first step refines *skip*.

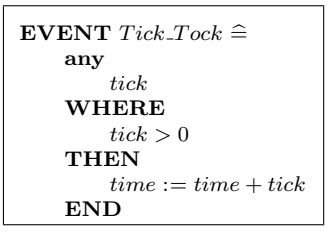
Now, in order to respond to the trigger event, two steps have to be accomplished, where the first step has an expiry and the second one has a deadline. In the concrete level, the trigger event of the expiry property for event B_1 is event A, and the trigger event for the deadline on event B_2, is event B_1. Hence, the abstract expiry is broken into a concrete expiry, and a concrete deadline, as shown in Figure 4.9(b), in such a way that their sequence, does not violate the abstract expiry ($t_1 + t_2 \leq t$).

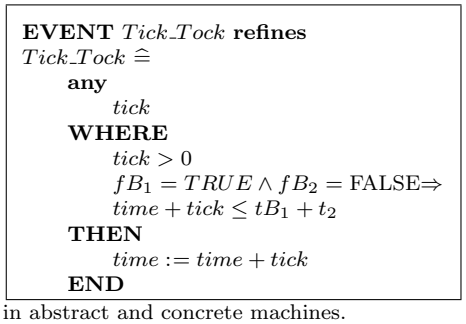


(a) Events A and B



(b) Events A, B_1 and B_2





(c) Event *Tick_Tock* in abstract and concrete machines.

Figure 4.9: Events A and B plus their expiry property in the abstract Machine in 4.9(a), followed by event A , events B_1 and B_2 in the concrete machine plus their concrete timing properties in 4.9(b).

We need to prove that the concrete machine refines the behaviour of its abstract one. For the refinement pattern, presented in Figure 4.8, it is necessary to prove the abstract expiry holds in the concrete machine.

As shown in Figure 4.9, in the concrete machine, the abstract expiry between event A and event B is refined by the following timing properties:

$$Expiry\ (A,\ B_1,\ t_1)\,, \tag{4.8a}$$

$$Deadline\ (B_1,\ B_2,\ t_2)\,. \tag{4.8b}$$

Based on deadline (4.8b) if event B_1 has happened then event B_2 has to happen within t_2 time-units. By having this timing property, the relation between occurrence times of event B_1 and event B_2, has been specified. But we need to specify the relation between the occurrence time of event B_2, and event A occurrence time. This can be achieved, by specifying the relation between the occurrence times of event B_1 and event A.

On the other hand, based on expiry (4.8a) we know that if B_1 does not happen before it expires ($tA + t_1$), then B_2 will never be enabled, since its trigger event (B_1) cannot happen. So event A in the concrete machine, triggers an expiry which may eventually cause event B_2 to never become enabled.

Two kinds of gluing invariants are needed, in order to prove that the concrete expiry and the concrete deadline, satisfy their abstract expiry. One to specify the relation of the abstract and the concrete events' occurrences, and another to specify the relation between the occurrence times of the concrete trigger and response events, with their abstract event's occurrence times. The ordering invariants are the same as the invariants presented in the refinement pattern of Section 4.3.1. The timing gluing invariants for the refinement pattern, presented in Figure 4.8, are as follows:

$$fB_1 \ = \ TRUE \ \Rightarrow \ tB_1 \ \leq \ tA \ + \ t_1, \tag{4.9}$$

The relation of the occurrence time of event B_1 and event A has been specified by the gluing invariant (4.9) which is part of the expiry semantics as explained in Section 4.2.2. Based on expiry (4.8a), it is guaranteed that if event B_1 has happened it was within t_1 time-units of event A occurrence. As a result, based on the deadline and expiry, the occurrence time of event B_2 has a following relation with occurrence time of event A:

$$fB_2 \ = TRUE \ \Rightarrow \ tB_2 \ \leq \ (tA \ + \ t_1) \ + \ t_2, \tag{4.10}$$

So, if B_1 happens before its expiry, it is guaranteed that B_2 will happen within $t_1 + t_2$ time-units of event A's occurrence. Since $t_1 + t_2 \leq t$ the timing refinement is consistent.

In an Event-B model of this refinement, 10 POs have been generated for the abstract machine which all were automatically. For the concrete machine, 25 POs have been generated from which only one has been discharged interactively.

4.3.3 Refining a Response Event of a Deadline by Several Alternative Responses

In a stepwise modelling process, sometimes, it is useful to generalize, the possible responses of a request, as a single response event, in order to express, and verify, their

common properties in the abstraction, and then talk about their exclusive properties, in a more concrete model, where they have been distinguished.

For instance, consider a case, where instead of refining event B, in the refinement pattern of Section 4.3.1, by two sequential sub-steps, it has been refined by two alternative events, B_1 and B_2, as shown in Figure 4.10. Event B_1 represents the main response scenario, and event B_2 represents the alternative one.

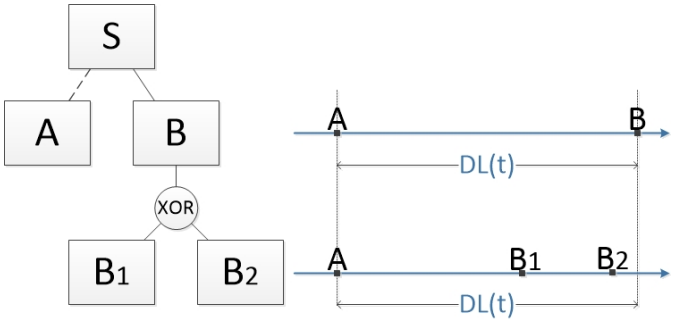


Figure 4.10: Refining an event by two alternative events. XOR in the refinement diagram represents the fact that either of B_1's occurrence, or B_2's occurrence in the refinement, is equivalent to the occurrence of event B in the abstract.

Based on the refinement, either of event B_1's occurrence, or event B_2's occurrence, are equivalent to occurrence of the abstract event (B). So the abstract deadline between event A and event B, satisfies by the occurrence of either of the refining events, before the deadline.

As shown in Figure 4.11(b), the concrete deadline is based on event A, as its trigger event, and either of event B_1, or event B_2, as the response events. The concrete deadline duration, is the same as its abstract deadline.

The only kind of invariant required to discharge refinement proof obligations of the timing property in this pattern, specifies the relation between the occurrences of the abstract and the concrete response events. For this generic refinement pattern, this invariant will be as follows:

$$fB_1 = TRUE \lor fB_2 = TRUE \Leftrightarrow fB = TRUE\,. \tag{4.11}$$

Based on invariant (4.11) either of event B_1 or event B_2 occurrences, is equivalent to the occurrence of event B. This invariant is required for control flow refinement.

In an Event-B model of this refinement, 14 POs have been generated for the abstract machine which all were automatically. For the concrete machine, 25 POs have been generated which all proved automatically.

TIMING
$Deadline(A, B, t)$
EVENT $A \widehat{=}$
WHERE
$fA = FALSE$
$G_A(c, v)$
THEN
$fA := TRUE$
Act_A
END
EVENT $B \widehat{=}$
WHERE
$fA = TRUE$
$fB = FALSE$
$G_B(c, v)$
THEN
$fB := TRUE$
Act_B
END

(a) Events A and B and their timing property.

TIMING
$Deadline(A, B_1 \vee B_2, t)$
EVENT $A \widehat{=}$
WHERE
$fA = FALSE$
$G_A(c, v)$
THEN
$fA := TRUE$
Act_A
END
EVENT B_1 *refines* $B \widehat{=}$
WHERE
$fA = TRUE$
$fB_1 = FALSE$
$fB_2 = FALSE$
$G_{B_1}(c, v)$
THEN
$fB_1 := TRUE$
Act_{B_1}
END
EVENT B_2 *refines* $B \widehat{=}$
WHERE
$fA = TRUE$
$fB_1 = FALSE$
$fB_2 = FALSE$
$G_{B_2}(c, v)$
THEN
$fB_2 := TRUE$
Act_{B_2}
END

(b) Events A, B_1 and B_2 and their timing property

EVENT $Tick\_Tock \widehat{=}$
any
$tick$
WHERE
$tick > 0$
$fA = TRUE \wedge fB = \text{FALSE} \Rightarrow$
$time + tick \leq tA + t$
THEN
$time := time + tick$
END

EVENT $Tick\_Tock$ **refines** $Tick\_Tock \widehat{=}$
any
$tick$
WHERE
$tick > 0$
$fA = TRUE \wedge fB_1 = \text{FALSE} \wedge fB_2 = \text{FALSE} \Rightarrow$
$time + tick \leq tA + t$
THEN
$time := time + tick$
END

(c) Event $Tick\_Tock$ in abstract and concrete machines.

Figure 4.11: Refining a trigger-response pattern and its timing properties to two alternative responses plus the concrete timing property.

4.3.4 Refining An Abstract Deadline to Alternative Sub-deadlines

Based on the same principles, mentioned in Section 4.3.3, it may also be useful to generalize, several alternative request-response sequences of a system, as a single request-response sequence. In this way, the general and exclusive properties of those sequences, can be verified in different levels of abstraction.

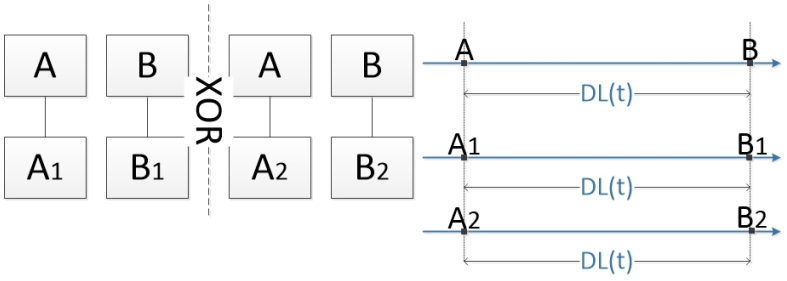
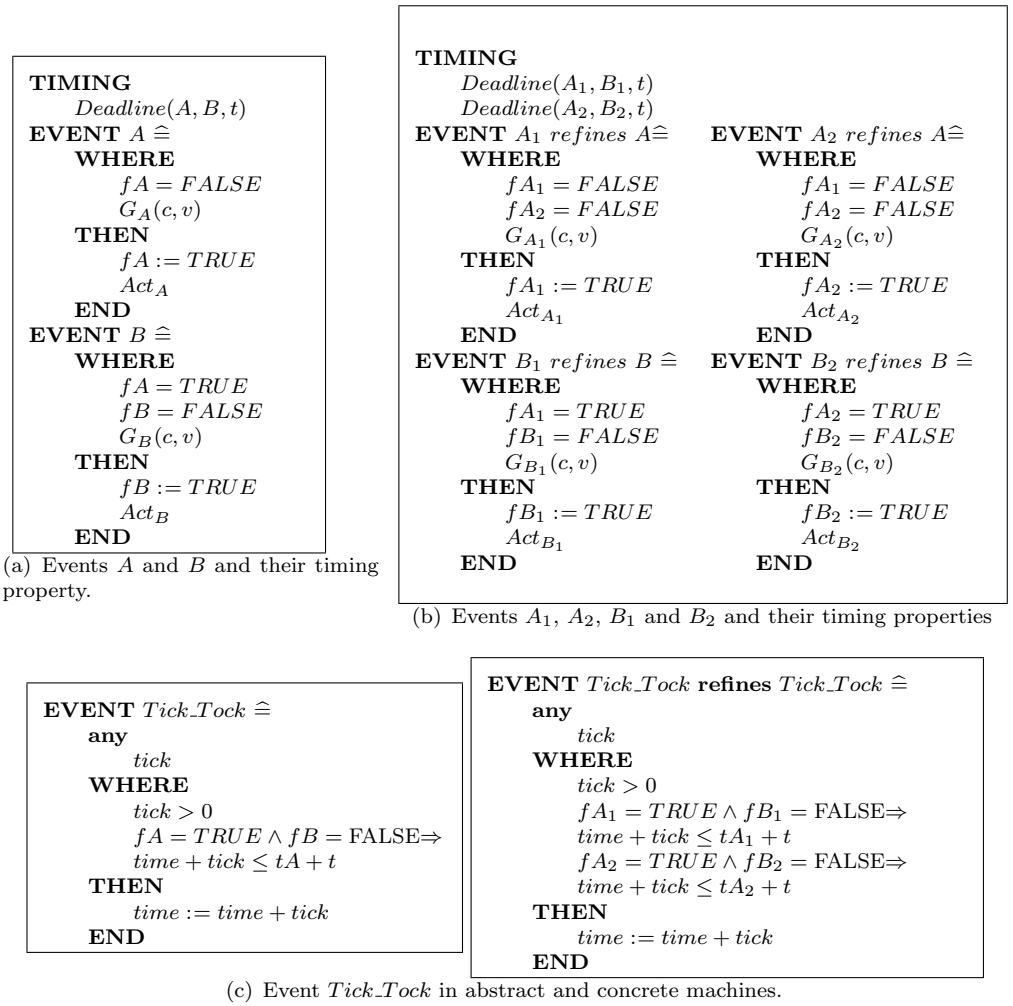


Figure 4.12: How a single trigger-response sequence can be refined to several trigger-response cases. XOR in the refinement diagram, represents the fact that the occurrence of either of those sequences, in the concrete level, is equivalent to the occurrence of the abstract sequence.

The difference between this case and the refinement pattern, explained in Section 4.3.3, is just about the trigger event. In Section 4.3.3 the trigger event has not been refined, but in this case the trigger event has been refined to several alternative cases, and each concrete trigger event is related to one of the alternative responses.

Consider the generic refinement pattern of Section 4.3.3, where also event A has been refined by two alternative events A_1 and A_2, as shown in Figure 4.12. In this refinement, event A_1 triggers event B_1 and event A_2 triggers event B_2. As a result, the abstract deadline between event A and event B, should be refined by the timing properties of the concrete alternative trigger-response sequences.



(a) Events A and B and their timing property.

(b) Events A_1, A_2, B_1 and B_2 and their timing properties

(c) Event $Tick\_Tock$ in abstract and concrete machines.

Figure 4.13: Refining a trigger-response pattern and its timing property, by two alternative trigger-response cases, and their corresponding concrete timing properties.

The only difference between the concrete deadlines and the abstract one, is the name of trigger and response events. As shown in Figure 4.13(b), there is a concrete deadline for each concrete trigger-response sequence, with a same duration as the abstract deadline.

To discharge the refinement proof obligations of the presented timing properties, two types of gluing invariants are required for each concrete trigger-response sequence:

1. Invariants to specify the relation between occurrences of the abstract events, and their concrete ones, which are required for the control flow refinement (required in the untimed model),
2. Invariants to specify the relation between the abstract trigger event's occurrence time and the occurrence times of the alternative concrete trigger events.

Based on our generic refinement pattern the required invariants should be as follows:

$$fA_1 = TRUE \vee fA_2 = TRUE \Leftrightarrow fA = TRUE \ (Type\ 1)\,, \tag{4.12a}$$

$$fA_1 = TRUE \Rightarrow fA_2 = FALSE \ (Type\ 1)\,, \tag{4.12b}$$

$$fB_1 = TRUE \vee fB_2 = TRUE \Leftrightarrow fB = TRUE \ (Type\ 1)\,, \tag{4.12c}$$

$$fA_1 = TRUE \Rightarrow tA_1 = tA \ (Type\ 2)\,, \tag{4.12d}$$

$$fA_2 = TRUE \Rightarrow tA_2 = tA \ (Type\ 2)\,. \tag{4.12e}$$

In an Event-B model of this refinement, 14 POs have been generated for the abstract machine which all were automatically. For the concrete machine, 48 POs have been generated from which only one required to be proved interactively.

4.3.5 Asymmetric Alternatives

Developing this pattern was triggered, when we were working on an automatic gear controller case study. Based on the case study, there were two conditions, assumed for the system, normal and difficult. After receiving a gear-changing request, controller tries to establish a synchronized speed between the engine and the gearbox, in order to release the currently engaged gear. But in a difficult situation, it is not possible to accomplish the first step in time, so after struggling to achieve the synchronized speed for a while, the controller will try to open the clutch instead, and then release the current gear by an open clutch.

In order to explain this refinement pattern, the example of Section 4.3.3, will be continued. In the current state, we have a trigger event A, and two alternative responses, events B_1 and B_2, which have replaced the abstract response event (event B). These two levels of abstraction have been constrained by some deadlines, presented in Figure 4.11.

In the next refinement, each of event B_1 and event B_2, have been replaced by two sequential sub-steps. Accordingly, the deadline of each alternative abstract response, will

be replaced by two sequential sub-deadlines, in a same way, presented in Section 4.3.1 (event B_1 will be replaced by events B_3 and B_4 sequence, and event B_2 will be replaced by events B_5 and B_6 sequence).

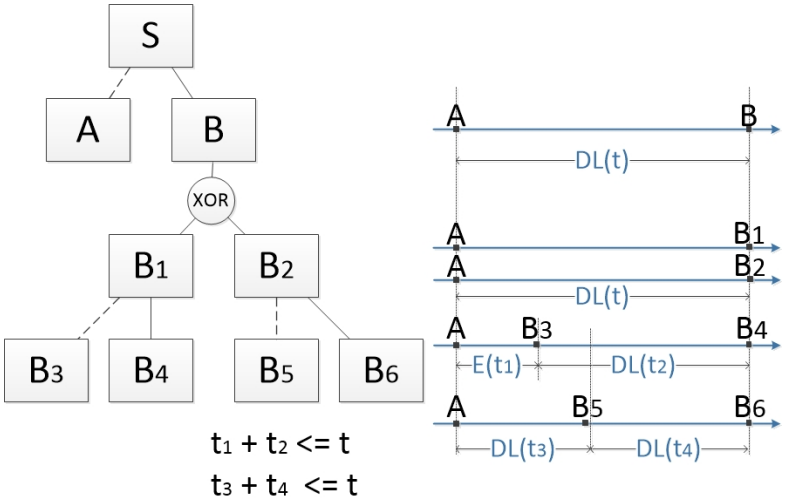


Figure 4.14: Refining each alternative response, by a sequence of two sub-steps. In this diagram $DL(t_3)$ constraints events B_3 and B_5, $DL(t_4)$ constraints event B_6, $DL(t_2)$ constraints event B_4, and $E(t_1)$ constraints event B_3.

In this refinement, an occurrence of A will be followed by one of its abstract responses (B_1 or B_2), as follows:

- The first step of B_1, represented by B_3, can only occur within time t_1 of A, and then its second step, B_4 occurs within time t_2 of B_3,
- if B_3 does not occur within time t_1 of A, instead the first step of B_2, represented by B_5, must occur within time t_3 of A, and then its following step (B_6), must occur within time t_4 of B_5 occurrence.

As a result, within t_3 time-units of the A's occurrence, either the first response case has been activated, or the second one has been activated (by the occurrence of their first steps).

The challenging timing property we want to model in this level, is that either B_3 has to happen by t_1 or B_5, have to happen by t_3. Based on the type of timing properties introduced in this work, this property can only be modelled, by a deadline and an expiry.

The main reason for having the expiry, is that we are not necessarily assuming that, the sequence of the concrete deadlines, between event A, event B_3, and B_4, satisfies the abstract deadline between event A and event B ($t_3 + t_2 \ \leq t \vee t_3 + t_2 \ > t$). Based on the expiry on event B_3, after a specific time, it cannot happen anymore, and the only possible response will be the alternative one, modelled by events B_5 and B_6.

TIMING
$Expiry(A, B_3, t_1)$
$Deadline(B_3, B_4, t_2)$
$Deadline(A, B_3 \vee B_5, t_3)$
$Deadline(B_5, B_6, t4)$

EVENT $B_3 \widehat{=}$
WHERE
$fA = TRUE$
$fB_3 = FALSE$
$fB_5 = FALSE$
$G_{B_3}(c, v)$
THEN
$fB_3 := TRUE$
Act_{B_3}
END

EVENT $B_5 \widehat{=}$
WHERE
$fA = TRUE$
$fB_3 = FALSE$
$fB_5 = FALSE$
$G_{B_5}(c, v)$
THEN
$fB_5 := TRUE$
Act_{B_5}
END

EVENT B_4 *refines* $B_1 \widehat{=}$
WHERE
$fA = TRUE$
$fB_3 = TRUE$
$fB_4 = FALSE$
$G_{B_4}(c, v)$
THEN
$fB_4 := TRUE$
Act_{B_4}
END

EVENT B_6 *refines* $B_2 \widehat{=}$
WHERE
$fA = TRUE$
$fB_5 = TRUE$
$fB_6 = FALSE$
$G_{B_6}(c, v)$
THEN
$fB_6 := TRUE$
Act_{B_6}
END

(a) Events B_3, B_4, B_5 and B_6.

EVENT $Tick\_Tock$ **refines** $Tick\_Tock \widehat{=}$
any
$tick$
WHERE
$tick > 0$
$fA = TRUE \wedge fB_1 = \text{FALSE} \wedge$
$fB_2 = \text{FALSE} \Rightarrow$
$time + tick \leq tA + t$
THEN
$time := time + tick$
END

EVENT $Tick\_Tock$ **refines** $Tick\_Tock \widehat{=}$
any
$tick$
WHERE
$tick > 0$
$fA = TRUE \wedge fB_3 = \text{FALSE} \wedge$
$fB_5 = \text{FALSE} \Rightarrow$
$time + tick \leq tA + t_3$
$fB_3 = TRUE \wedge fB_4 = \text{FALSE} \Rightarrow$
$time + tick \leq tB_3 + t_2$
$fB_5 = TRUE \wedge fB_6 = \text{FALSE} \Rightarrow$
$time + tick \leq tB_5 + t_4$
THEN
$time := time + tick$
END

(b) Event $Tick\_Tock$ in the second and third levels of abstractions.

Figure 4.15: Events B_3, B_4, B_5 and B_6 of the most concrete model, and their timing properties. Plus the $Tick\_Tock$ event in the most concrete and its abstract machines.

For example, imagine in a car, there are two possible scenarios to release the currently engaged gear, scenarios R_1 and R_2, and each of them consists of two sequential steps. Whichever scenario that its first step happens first, will be the one the system goes with. Besides there is a deadline to force one and only one of these two first steps to happen by at most t_2 of triggering the releasing process. The second steps are constrained by deadlines from the occurrence of their corresponding first step. But in scenario R_1 the sequence of its first and second steps' deadlines $(t_1 + t_2)$ is greater than the deadline of the whole releasing process (t). Instead its first step has an expiry (t_1) which followed by the deadline of the second step (t_2), satisfies the deadline of the whole process. As a result, if the first step of scenario R_1 does not happen by its expiry, the only possible scenario of releasing the currently engaged gear will be R_2.

By enforcing this property with an expiry as shown in Figure 4.15, the concrete timing properties will satisfy their abstract ones. By having an expiry property, between events A and B_3, for a period of t_1, it will be guaranteed that if event B_3 has occurred, it was within time t_1 of A occurrence (Formulated in invariant (4.13)).

$$fB_3 = TRUE \Rightarrow tB_3 \leq tA + t_1 \,. \tag{4.13}$$

From event B_3 occurrence, event B_4 has t_2 time-units, to happen based on the existing deadline between them. As a result, the abstract deadline has been contained, by the concrete timing properties. Invariant (4.13), plus the invariants, presented in Section 4.3.1, will be required to prove the consistency of the refinement. This pattern shows how combination of deadline and expiry can be used to model timing properties of a time-critical system.

To prove the correctness preservation by the last refinement, we need the following invariants other than those included in the semantics of deadline and expiry:

$$fB_4 = fB_1 \,, \tag{4.14a}$$

$$fB_6 = fB_2 \,, \tag{4.14b}$$

$$fB_3 = TRUE \Rightarrow fB_5 = FALSE \,. \tag{4.14c}$$

These invariants correspond to the control flow refinement, so they will be required in the untimed model too. Invariant 4.14c express the property that only one of the alternative scenarios can be activated. As a result, one and only one of the first steps has to happen.

Some combinations of timing properties my cause deadlock, or disable an event indefinitely. For example, in the refinement pattern explained in this section, we have a combination of expiry and deadlines, which may cause the $Tick\_Tock$ event to be disabled forever. The effects of timing properties on events' enableness and how they can cause a deadlock in an Event-B model, will be discussed in chapter 6.

In an Event-B model of this refinement, 14 POs have been generated for the abstract machine which all were proved automatically. For the second level of abstraction, 25 POs have been generated, and all of them were discharged automatically. For the most concrete machine, 59 POs have been generated from which only one required to be proved interactively.

4.3.5.1 Disjunctive Deadlines vs. Deadline and Expiry Combination

The reader may ask, why we do not use two disjunctive deadlines, instead of a deadline and an expiry, for timing properties similar to events A, B_3, and B_4, in the example

of Section 4.3.5. To explain why two disjunctive deadlines do not enforce the desirable behaviour, in this section, we will go through the refinement process of an abstract deadline by two disjunctive deadlines, on a same control flow refinement pattern.

Suppose we want to encode timing properties of events A, B_3 and B_5, as follows:

$$Deadline(A, B_3, t_1) \vee Deadline(A, B_5, t_3) \tag{4.15a}$$

Where

$$t_1 < t_3$$

To encode these properties in an Event-B model, the required guard on the $Tick\_Tock$ event will be as follows, based on the introduced deadline semantics in Section 4.2.3:

$$\begin{gathered}(fA = TRUE \wedge fB_3 = FALSE \Rightarrow time + tick \leq tA + t_1) \\ \vee \\ (fA = TRUE \wedge fB_5 = FALSE \Rightarrow time + tick \leq tA + t_3)\end{gathered} \tag{4.16}$$

A deadline of t_3, between event A, as the trigger, and events B_3 and B_5, as its responses, as follows:

$$Deadline\ (A,\ B_3\ \vee\ B_5,\ t_3)\,, \tag{4.17}$$

Will enforce the following guard, on the $Tick\_Tock$ event, based on the deadline semantics, explained in Section 4.2.3:

$$fA\ = TRUE \wedge fB_3\ = FALSE \wedge fB_5\ = FALSE \Rightarrow time + tick \leq tA + t_3\,. \tag{4.18}$$

It is easy to prove that guard (4.17), is logically equivalent to guard (4.18), since $t_1 < t_3$. Accordingly, disjunctive deadlines (4.15a), allow event B_3 to occur after time t_1 of A occurrence. As a result, by having two disjunctive deadlines, the expiry property on event B_3 will not be enforced.

In this section some approaches have been introduced in order to refine timing properties, based on several generic control flow refinement patterns. These patterns do not contain all the possible cases of refining timing properties, and other possible refinement patterns are part of the future works.

4.4 Alternative Ways of Encoding a Sequential Control Flow in Event-B

As explained in Section 4.1, our approach to model timing properties in Event-B, is based on the assumption, that sequential control flow is encoded by using boolean flags.

This assumption is the result of our investigation on existing approaches to model a sequential order in an Event-B model.

The control flow can be presented as sequences of events, or as sequences of states. Since, Event-B, is an event based method, it is natural to model the control flow, based on the order of events.

Other than using boolean flags to model the sequential order between trigger events and their response events, it can be modelled by a set too. It is possible to dedicate a constant to each event, and declare a set, which is empty in the initialization, and the occurrence of each event causes its corresponding constant, to be added to that set, as demonstrated for an order between event A and event B, in Figure 4.16(b). Based on this approach, if event B has to happen after event A, then event B has to check if the corresponding constant of event A, has been already added to the set.

EVENT $A \widehat{=}$
  WHERE
    $A = FALSE$
    $G_A(c, v)$
  THEN
    $A := TRUE$
    Act_A
  END
EVENT $B \widehat{=}$
  WHERE
    $A = TRUE$
    $B = FALSE$
    $G_B(c, v)$
  THEN
    $B := TRUE$
    Act_B
  END

(a) Modelling a sequential order by using boolean flags.

EVENT $A \widehat{=}$
  WHERE
    $A \notin order$
    $G_A(c, v)$
  THEN
    $order := order \cup \{A\}$
    Act_A
  END
EVENT $B \widehat{=}$
  WHERE
    $A \in order$
    $B \notin order$
    $G_B(c, v)$
  THEN
    $order := order \cup \{B\}$
    Act_B
  END

(b) Modelling a sequential order by using a set.

Figure 4.16: Two existing approaches to model a sequential order has been shown for an order between generic events A and B

This approach becomes problematic, during horizontal refinements. If a *skip* event is added in a refinement, a new set has to be declared, because it is not possible to manipulate an abstract variable in a *skip* event. As shown in Figure 4.17(b) because of adding a new event ($preB$), between events A and B, in the refinement, the previous event occurrence history set ($order$), has been replaced by a new one ($newOrder$).

Declaring a new ordering set, requires the modeller to provide the relation between the abstract ordering set, and the concrete one, for all the possible members of these two sets. Based on the extent of a refinement, and the number of existing events in the corresponding machine, this process can be time consuming and impractical.

For a control flow refinement pattern, where the order between events A and B, has been refined to an order between events A, $preB$ and B, the following gluing invariants

EVENT A *refines* $A \widehat{=}$
  WHERE
    $A = FALSE$
    $G_A(c, v)$
  THEN
    $A := TRUE$
    Act_A
  END
EVENT $preB \widehat{=}$
  WHERE
    $A = TRUE$
    $preB = FALSE$
    $G_{preB}(c, v)$
  THEN
    $preB := TRUE$
    Act_{preB}
  END
EVENT B *refines* $B \widehat{=}$
  WHERE
    $preB = TRUE$
    $B = FALSE$
    $G_B(c, v)$
  THEN
    $B := TRUE$
    Act_B
  END

(a) Refining a sequential order modelled by boolean flags.

EVENT A *refines* $A \widehat{=}$
  WHERE
    $A \notin newOrder$
    $G_A(c, v)$
  THEN
    $newOrder := newOrder \cup \{A\}$
    Act_A
  END
EVENT $preB \widehat{=}$
  WHERE
    $A \in newOrder$
    $preB \notin newOrder$
    $G_{preB}(c, v)$
  THEN
    $newOrder := newOrder \cup \{preB\}$
    Act_{preB}
  END
EVENT B *refines* $B \widehat{=}$
  WHERE
    $preB \in newOrder$
    $B \notin newOrder$
    $G_B(c, v)$
  THEN
    $newOrder := newOrder \cup \{B\}$
    Act_B
  END

(b) Refining a sequential order modelled by a set.

Figure 4.17: Effects of adding a *skip* event in a refinement, to a sequential order, modelled by using an occurrence history set.

are required, plus the type invariants of the new ordering set ($newOrder$):

$$newOrder \setminus \{preB\} \subseteq order\,, \tag{4.19a}$$

$$preB \in newOrder \Rightarrow A \in newOrder\,. \tag{4.19b}$$

Invariant 4.19a specifies the relation between occurrence of the refining events, and their abstracts. Whereas invariant 4.19b specifies the order between an occurrence of new event ($preB$), and an occurrence of event A. Besides, modeller needs to replace the abstract ordering set with the new one in the guards and actions of all the refining events.

On the other hand, based on the boolean flags approach, each flag is independent of the others, and adding a *skip* event, or refining an existing one, does not affect others.

By adding a new event, only the immediate neighbours of the new event, in the control flow sequence, will be affected, since their guard should be changed based on the concrete order. For the example of Figure 4.17(a), the only required gluing invariant, is as follows:

$$preB = TRUE \Rightarrow A = TRUE \tag{4.20}$$

As shown in these two examples, using a set, in order to model the sequential control flow of a model, instead of boolean flags, makes horizontal refinements more complex, since it requires more gluing invariants, to prove the consistency of a refinement.

Causing fewer changes, during horizontal refinements (since a boolean flag is just shared between immediate neighbours of a control flow sequence), and requiring fewer gluing invariants, in order to prove the consistency of a control flow refinement, have convinced us to assume this approach to encoding timing properties in an Event-B model.

4.5 Achievements

In this chapter three groups of timing properties have been defined, and some constructs have been introduced to express them. Then the semantics of those timing properties have been defined, and some patterns to refine them, based on some generic control flow refinement patterns, have been explained.

In Section 3.12 some of the existing works on modelling real-time systems in Event-B have been mentioned. Now, based on what has been presented in this chapter, it is possible to compare them to our approach. The downside of Cansell's [44] approach is that it only covers timing properties that force an event to happen in a specific time. So it is not possible to model different varieties of timing properties (e.g. delay and expiry).

In Bryans's [37] case study, the value of the activation set for each message acts as an expiry constraint for the event which represent the normal scenario, and acts as a delay for the recovery scenario. In the receive on-time event, the guard disables the event if the value of the current time is greater than the corresponding added value, which represents the receiving time. On the other hand, the guard of the receive late event disables it if the current time is less than the added activation time for the corresponding message. Although this approach covers more timing properties than the Cansell's [44] approach, but it is not able to model timing properties, which force events to occur before specific times (Deadline in our approach). As mentioned in Section 3.1, having hard deadlines distinguishes safety critical real-time systems. As a result, this approach cannot be used for modelling and verification of safety critical real-time systems.

None of these works tackle the refinement or decomposition of timing properties. In Cansell [44] and Bryans [37] works, time and timing properties have been added to a model as a refinement, but refining the timing properties have not been investigated.

Even though they have not examine the refinement of timing properties, we modelled several simply case-studies to evaluate the refinement process of timing properties, modelled in these approaches. Based on Cansell's [44] approach, since a timing property is modelled by adding and removing an integer to an activation set, after refining it by several concrete timing properties, in order to discharge the refinement consistency POs, we have to prove the following properties of the refinement:

- If the abstract activation set is not empty, then the concrete one is not empty,

- If the abstract and the concrete activation sets are not empty, then the minimum of the concrete set, is less than or equal to the minimum of the abstract one.

Proving these properties is challenging, and requires several gluing invariants, and some of them are hard to be specified in a complex system. To demonstrate the problem, we will go through the refinement of a timing property by two sequential sub-timing properties in the following.

Assume a trigger event A and its response event B, where B has to happen 10 times-units after occurrence of A. As shown in Figure 4.18, B has been refined by two sequential sub-responses, B_1 and B_2, where B_1 has to happen 5 time-units after A, and B_2 has to happen 5 time-units after B_1.

EVENT $A \widehat{=}$
WHERE
$A = FALSE$
THEN
$A := TRUE$
$tA := time$
$at := at \cup \{time + 10\}$
END
EVENT $B \widehat{=}$
WHERE
$A = TRUE$
$B = FALSE$
$time = tA + 10$
THEN
$B := TRUE$
$at := at \setminus \{time\}$
END

EVENT $Tick\_Tock \widehat{=}$
any
tm
WHERE
$tm > time$
$at \neq \emptyset \Rightarrow tm \leq min(at)$
THEN
$time := tm$
END

(a) Events A and B plus their timing properties

EVENT $A \widehat{=}$
WHERE
$A = FALSE$
THEN
$A := TRUE$
$tA := time$
$cat := cat \cup \{time + 5\}$
END
EVENT $B_1 \widehat{=}$
WHERE
$A = TRUE$
$B_1 = FALSE$
$time = tA + 5$
THEN
$B_1 := TRUE$
$tB_1 := time$
$cat := (cat \setminus \{time\}) \cup \{time + 5\}$
END

EVENT $B_2 \widehat{=}$
WHERE
$B_1 = TRUE$
$B_2 = FALSE$
$time = tB_1 + 5$
THEN
$B_2 := TRUE$
$cat := cat \setminus \{time\}$
END
EVENT $Tick\_Tock \widehat{=}$
any
tm
WHERE
$tm > time$
$cat \neq \emptyset \Rightarrow tm \leq min(cat)$
THEN
$time := tm$
END

(b) The concrete machine.

Figure 4.18: Refining a timing properties to two sequential sub-timing properties, where timing properties have been enforced based on Cansell's [44] approach.

As shown in Figure 4.18(b), the abstract activation-times' set *ac*, has been refined by *cat*. As a result, the following gluing invariants are required to prove the consistency of

the timing refinement:

$$at \neq \emptyset \Rightarrow Cat \neq \emptyset\,, \tag{4.21a}$$

$$Cat \neq \emptyset \wedge at \neq \emptyset \Rightarrow min(Cat) = min(at)\,. \tag{4.21b}$$

In order to discharge the corresponding POs of invariants (4.21a) and 4.21b, some invariants need to be added to the model. These additional invariants are as follows:

$$A = FALSE \Rightarrow at = \emptyset\,, \tag{4.22a}$$

$$B = TRUE \Rightarrow at = \emptyset\,, \tag{4.22b}$$

$$A = TRUE \wedge B = FALSE \Rightarrow at = \{tA + 10\}\,. \tag{4.22c}$$

Proving the added invariants, to discharge the POs of invariant (4.21b) is not a straightforward process. The problematic invariant is (4.22c). In order to prove that the minimum of the concrete activation-times is always less than or equal to the abstract one, we need to specify the exact state of the abstract set, within the duration between the occurrences of the abstract trigger and response events.

Using the same variable to model all the timing properties of a system has made the invariants' proving process more complex. On the other hand, in our approach, each timing property has its own variables, which are only manipulated by the corresponding events of that property. As explained in [42], keeping separate structures separate eases the proof effort.

Assume a model where there are some events which their occurrences change the set of activation-times, and they can happen between the abstract trigger and response events non-deterministically, number of possible states for the activation set in this period will depend on how many of those events exist in the model, and how many alternative sequences can be assumed for their occurrences.

In Sections 3.5 and 3.9, real-timed CSP and VDM were introduced. Their downside in comparison to our approach is the lack of tool support for the refinement of timing properties.

Besides all of these related works, one possible alteration to the approach introduced in this chapter, is to use the occurrence time variables, instead of boolean flags, to check whether an event has happened or not. In this way, we set the current time value to 1 in the initialization, and all the occurrence time variables to 0. As a result, time starts from 1 in the model and goes up, and if an event has not happened yet, its occurrence time value is 0, otherwise it is greater than 0. For example, how the delay semantics will be changed based on this approach has been presented in Figure 4.19.

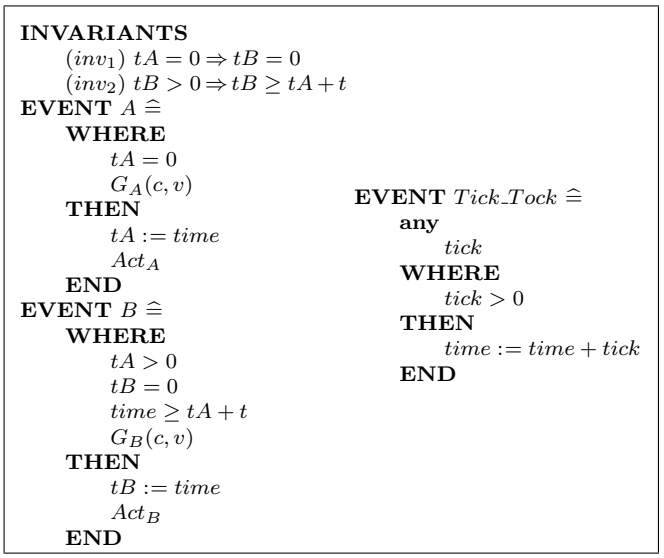


Figure 4.19: Semantic of a delay property in Event-B, where the occurrence time variables have been used to detect events' occurrences.

Since the variable which records the occurrence time of an event is going to be used to check whether it has happened or not, to model control loops, by the end of each iteration, all the occurrence time variables should be set to 0. The main advantage of this approach, is that the semantics of timing properties is based on fewer variables, which makes it much simpler. This alteration can be investigated as part of future works.

In the following Chapter, how a timed Event-B model can be decomposed, and some of the challenges, we were facing during its application, in our case-studies, will be explained.

Chapter 5

Decomposition of Timed Event-B Models

As mentioned in Section 2.8.5, one of the approaches to dealing with the complexity of a system model is *decomposition*. In this work, we have explored the use of shared-event decomposition to decompose timed Event-B models, which lets us refine the timing properties of the resulting sub-components independently.

By decomposing a timed Event-B model, to two sub-components, one representing the environment, and the other the controller, we will be able to refine the timing properties of the controller independent from the environment. Since the timing properties will be decomposed too, the $Tick\_Tock$ event of the controller, will be much simpler (deadlines' guards are decomposed). As a result, deadlines' refinements will be less complex to perform. For example, in the automatic gear controller case study, explained in Chapter 7, Before decomposition, we had 54 deadline guards in the $Tick\_Tock$ event, but after decomposing the model to the controller and its environment, the $Tick\_Tock$ event of the controller had 21 deadline guards.

Treating both shared variables and shared events, is important in timed Event-B decomposition, because *time* is a shared variable and its progress is a shared event. What we want is a decomposition approach, in which timing properties of each components can be refined or enhanced independently.

In the following, we discuss how a timed Event-B model can be decomposed.

5.1 Timed Event-B Decomposition Process

In order to decompose a timed Event-B model, some changes, in terms of a refinement, are required to be applied. What we want to get out of the decomposition, is a set of sub-components' timed models, where the progress of time is synchronous, and each model can be refined independently, and it just includes its corresponding timing properties.

Since there should be no common state variable between the components of a shared-event decomposition, and the semantics of timing properties is based on state variables, which keep track of events' occurrences and their occurrence-times, a timed Event-B model has to have the following characteristics before the decomposition:

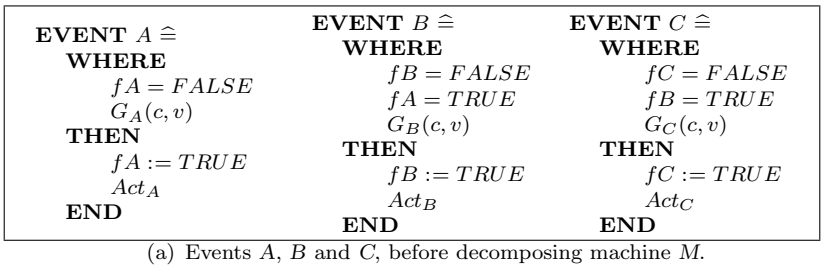
1. For each shared event, there should be a state variable in each component recording its occurrence,
2. For each shared event, involved in a timing property, there should be a state variable in each component recording its occurrence time,
3. The current time value, should be available in every component,
4. The corresponding component of each timing property, has to be specified,
5. Each timing property has to be encoded by using the variables of its corresponding component.

For items 1 and 2, the occurrence flag variable and the occurrence time variable of a shared event, involved in a timing property, should be replicated as many as the number of sub-components, sharing that event. Since these variables are modified only by the shared events, these replication will be a valid refinement. Based on the same principle, the *time* variable should be replicated for all the sub-components (item 3).

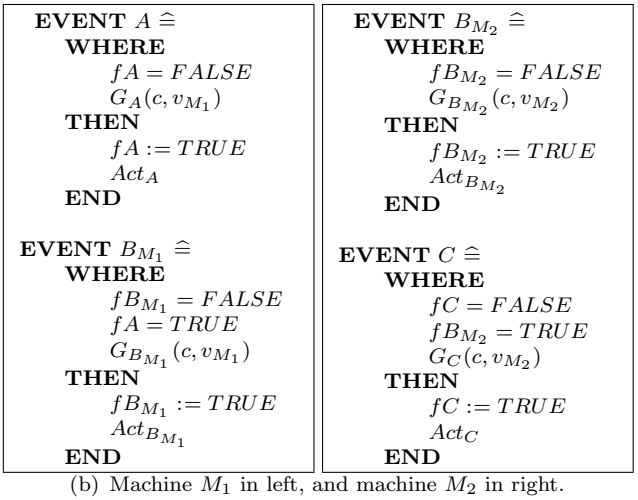
The owner of a timing property can be specified based on its trigger and response events. A timing property, belongs to the machine, containing all the events involved in that timing property (item 4).

Finally to satisfy item 5, in the encoding of each timing property, the replicas of the current time variable, which belongs to the owner of that timing property has to be used. Besides, if a shared event is involved in a timing property, the corresponding replica of its occurrence flag and occurrence time has to be used in the encoding of that timing property, according to the sub-component it belongs to.

To have a better understanding of the process, we will go through the decomposition process of a simple example in the following.



(a) Events A, B and C, before decomposing machine M.



(b) Machine M_1 in left, and machine M_2 in right.

Figure 5.1: Decomposing a machine with three events into two machines.

Assume a machine M with three events A, B and C. As shown in Figure 5.1(a), these event happen in the same order, they have been introduced. If two deadlines constrain the occurrence of these events as follows:

$$Deadline(A, B, x)\,, \tag{5.1a}$$

$$Deadline(B, C, y)\,. \tag{5.1b}$$

In order to decompose M into two machines M_1 and M_2, where A belongs to M_1, C belongs to M_2, and B is the shared event (Figure 5.1(b)), there should be two copies of the current time, the occurrence flag of event B, and its occurrence time variable (items 1, 2, and 3).

Based on this decomposition pattern, deadline (5.1a) is the timing property of M_1, because both A and B appear in M_1, and deadline (5.1b) belongs to M_2, since its events appear there (item 4). As mentioned before, the $Tick\_Tock$ event will be shared by all the sub-components.

According to the semantics of deadline, the corresponding guards of (5.1a) and (5.1b), before applying the changes required for a decomposition, are as follows:

$$fA = TRUE \wedge fB = FALSE \Rightarrow time + tick \leq tA + x\,, \tag{5.2a}$$

$$fB = TRUE \wedge fC = FALSE \Rightarrow time + tick \leq tB + y\,. \tag{5.2b}$$

After replicating the occurrence flags and occurrence times, based on the ownership of the timing properties, the deadlines guards will be changed as follows (item 5):

$$fA = TRUE \wedge fB_{M_1} = FALSE \Rightarrow time_{M_1} + tick \leq tA + x\,, \tag{5.3a}$$

$$fB_{M_2} = TRUE \wedge fC = FALSE \Rightarrow time_{M_2} + tick \leq tB_{M_2} + y\,. \tag{5.3b}$$

Where B_{M_1} and B_{M_2} are the replicas of event B's occurrence flag, and $B_{M_1}t$ and $B_{M_2}t$ are the replicas of its occurrence time. Besides, $time_{M_1}$ and $time_{M_2}$ are replicas of the current time for each component. By these changes, the $Tick\_Tock$ event becomes a parallel composition of two events:

$Tick\_Tock1 \parallel Tick\_Tock2 \mathrel{\widehat{=}}$
any $tick$
where
$fA = TRUE \wedge fB_{M_1} = FALSE \Rightarrow time_{M_1} + tick \leq tA + x$
$\wedge$
$fB_{M_2} = TRUE \wedge fC = FALSE \Rightarrow time_{M_2} + tick \leq tB_{M_2} + y$
then
$time_{M_1} := time_{M_1} + tick \parallel time_{M_2} := time_{M_2} + tick$
end.

Now this event can be decomposed into M_1 and M_2 where the corresponding guard of deadline (5.1a) will appear in the $Tick\_Tock$ of M_1, and the corresponding guard of deadline (5.1b) will appear in the $Tick\_Tock$ event of M_2.

5.2 The Challenge of Decomposing Timed Control Loops

After decomposing the timed Event-B model of the gear controller case study (Chapter 7), we found out that the approach we used to model a control loop has caused an undesirable behaviour in the decomposed machines. As mentioned before, in a sequential control flow, we use boolean flags, which keep track of events' occurrences. Consequently, in a control loop, after each iteration, these flags have to be reset for the next iteration. We introduced an event, named *FINAL*, which happens at the end of each iteration and resets all the flags.

Having the *FINAL* event, has a disadvantage. In the decomposition, *FINAL* will be shared by all the components, since it resets all the existing occurrence flags. Sharing *FINAL* event between all the components, means that their reset is synchronous, which is not necessarily the case.

This is caused because of the way the timing properties are encoded. We need a history of events' occurrences to check whether the trigger has happened or not, and if it

has, whether its response has happened in the specified period of time or not. These properties have been express through the invariants and guards included in the timing properties' semantics. In the controller, the environment's changes are tracked by some variables (e.g. boolean flags), but there is no occurrence flag in the real world. As explained in Section 2.8, in Event-B our intention of modelling is to formalized a system in which there is a certain piece of software (the final product), as well as its environment. Besides, as mentioned in Chapter 2, there is no semicolon in Event-B, like other formal languages such as GCL and action systems, to specify the order of events. As a result we need occurrence flags to express the order of environment's events in Event-B.

For example in the automatic gear controller case study (Chapter 7), in response to a gear change request the clutch may be opened to release the currently engaged gear. Based on the mechanic of the clutch, it takes some specific period of time for it to be opened. This continues process has to be modelled by two events in a discrete modelling enviornment such as Event-B, one representing the beginning of the process, and one the end of it. As a result, to model the order of events and their timing properties, there should be a mechanism to keep track of the occurred events during a gear changing process.

Based on what has been explained, the occurrence flags of the environment's events are the means of modelling. So the $FINAL$ event in an Event-B model of environment is not a representation of a real world event, and its synchronization with the $FINAL$ event in the controller, is just the consequence of how the order of events is modelled in Event-B.

But if we do not want to have this synchronization between the $FINAL$ events, breaking it before the decomposition, can solve the problem. In this approach, before decomposing the model, based on the propagation pattern of the events among the decomposed components, the *FINAL* event will be broken, in order to have an independent reset event for each component after the decomposition. To do that, the occurrence flags needs to be replicated. In the following, we will go through this process for the example of Figure 5.2.

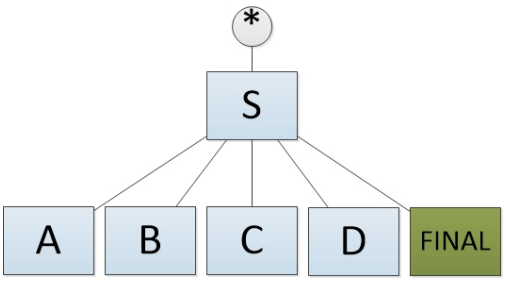


Figure 5.2: Sequence of four events in a loop, modelled by resetting the occurrence flags at the end of each iteration by occurrence of the *FINAL* event.

As shown in Figure 5.2, the most concrete model has five events, A, B, C, D and $FINAL$, which happen in the same order in each iteration. As explained before, based on the approach we use to model the control flow of the events, each of them will have a boolean occurrence flag (except the $FINAL$ event), which has to be reset by the $FINAL$ event at the end of each iteration (Figure 5.3).

EVENT $FINAL \widehat{=}$
  WHERE
    $fD = TRUE$
  THEN
    $fA := FALSE$
    $fB := FALSE$
    $fC := FALSE$
    $fD := FALSE$
  END

Figure 5.3: Semantic of a delay property in Event-B.

If we want to decompose the model into two components, M_1 and M_2, where event A belongs to M_1 and event C belongs to M_2, and events B and D are shared, the boolean flags will be refined as follows:

- Event A occurrence flag will be replaced by a new occurrence flag A_1,
- Event B occurrence flag will be replaced by two new occurrence flags B_1 and B_2,
- Event C occurrence flag will be replaced by a new occurrence flag C_2,
- Event D occurrence flag will be replaced by two new occurrence flags D_1 and D_2.

Based on these changes the reset event can be broken into two independent resetting events. As shown in Figure 5.4, event $FINAL1$ resets the occurrence flags of M_1's events, and event $FINAL2$ does it for the events of M_2.

EVENT $FINAL1$
refines $FINAL \widehat{=}$
  WHERE
    $fD1 = TRUE$
  THEN
    $fA1 := FALSE$
    $fB1 := FALSE$
    $fD1 := FALSE$
  END

EVENT $FINAL2 \widehat{=}$
  WHERE
    $fD2 = TRUE$
  THEN
    $fB2 := FALSE$
    $fC2 := FALSE$
    $fD2 := FALSE$
  END

Figure 5.4: Breaking the $FINAL$ event into two sub-resetting events.

In this example, we assumed that the abstract $FINAL$ event will be refined by $FINAL1$. Based on this refinement pattern, the required gluing invariants to prove its consistency

are as follows:

$$fA = fA1\,, \tag{5.4a}$$

$$fD = fD1\,, \tag{5.4b}$$

$$fB = fB1\,, \tag{5.4c}$$

$$fB2 = TRUE \wedge fC2 = FALSE \Rightarrow fB = TRUE\,, \tag{5.4d}$$

$$fD2 = TRUE \Rightarrow fC2 = TRUE\,, \tag{5.4e}$$

$$fD1 = TRUE \wedge fD2 = FALSE \Rightarrow fB2 = FALSE\,, \tag{5.4f}$$

$$fB2 = TRUE \wedge fC2 = FALSE \Rightarrow fC = FALSE\,, \tag{5.4g}$$

$$fC2 = TRUE \wedge fD2 = FALSE \Rightarrow fC = TRUE\,, \tag{5.4h}$$

$$fC2 = TRUE \Rightarrow fB2 = TRUE\,. \tag{5.4i}$$

Factors such as the *FINAL* event's refinement pattern, the control-flow of the events, and the decomposition pattern, affect the required gluing invariants to prove the consistency of breaking the *FINAL* event. Because of the number of factors influencing the process, it is not possible to introduce some specific patterns for the required gluing invariants, in order to mechanize this process.

For the Event-B model of the simple example we explained above, 20 POs were generated for the abstract machine, and 49 POs for the refining machine in the Rodin tool-set, which all have been discharged automatically. But the most challenging part was coming up with the gluing invariants, which was only possible by simulating the model, and tracing the changes of abstract flags in respect to their concrete ones.

Since breaking the *FINAL* event manually, is a complex process, which requires a considerable amount of effort for a real-size model, not being able to atomize it, is a considerable disadvantage.

In this chapter the required pre-steps of decomposing a timed Event-B model has been explained and demonstrated. Besides, the issue of decomposing a control loop was discussed and some possible solutions, accompanied by their advantages and disadvantages have been explained. By supporting the decomposition feature of Event-B by the introduced timing properties semantics, we can decomposed the controller from its environment, and then add the timing properties of its implementation independently. Decomposing the timing properties of a model, decreases the its complexity considerable, and makes it much easier to be refined. As mentioned before, usually we have deadlines and expiries in the more abstract models, and delays are introduced in the more concrete models, where there is no more horizontal refinement to do. Besides as it will be discussed in Chapter 6, composition of timing properties may cause deadlock. By decomposing the controller from its environment and then introducing the delay

properties, investigating the effects of timing properties on events' enableness will be less complex to perform.

In the next chapter, the effect of the timing properties of a machine, on its events' enableness will be investigated.

Chapter 6

Enableness of Response Events and the Tick_Tock Event

In this chapter, we introduce an approach to verify two important properties of a timed Event-B model. These two properties are as follows:

P1. Adding timing properties still allows a response event to occur, at least for a certain period of time,

P2. Time is not prevented from progressing indefinitely.

It is trivial to consider that an untimed Event-B model S is refined by its timed one as follows:

$$\begin{aligned} S &\sqsubseteq tS \\ skip &\sqsubseteq Tick\_Tock \end{aligned} \tag{6.1}$$

Where tS represents the events of S plus the corresponding guards and actions of its timing properties based on their semantics. So by checking the above properties for a timed Event-B model, we are actually verifying the enableness preservation in refinement, where the refinement is adding the timing properties. By adding timing properties we are strengthening the guards which can cause a deadlock.

6.1 Effects of Isolated Timing Properties

In order to verify P1 and P2, we have made another assumption about the enableness of a trigger-response pattern. We are assuming that in S (the untimed Event-B model of a system), if a trigger event has happened, at least one of its responses will be enabled eventually.

Figure 6.1: Indirect triggering state diagram.

Assume a trigger event A and its response event B. Either A enables B directly, in which B will be enabled immediately after the occurrence of A:

$$A \wedge \neg B \Rightarrow gd(B)\,, \tag{6.2}$$

Or the enabling relation between the trigger and its response events is transitive. In the later case, there are finite possible sequences of intermediate events, which eventually enable B, as shown in Figure 6.1.

So, for a set of intermediate events $C_1..C_n$, ensuring the eventual enabling of response event B can be formulated as follows:

$$A \wedge \neg B \Rightarrow gd(C_1) \vee .. \vee gd(C_n) \vee gd(B)\,, \tag{6.3a}$$

$$C_1..C_n \;\; are \;\; convergent\,. \tag{6.3b}$$

We are assuming that Predicate (6.3a) has already been verified, and the convergence of the intermediate events means that eventually there will be no enabled intermediate event. As a result, based on Predicates (6.3a) and (6.3b), after occurrence of A, if B has not happened yet, eventually B will become enabled

$$AG(A \rightarrow AF(gd(B)))\,. \tag{6.4}$$

More information about convergence in Event-B can be found in [13].

By this assumption, a deadline by itself does not violate any of P1 or P2. First of all, the added guard of a deadline, will be on the $Tick\_Tock$ event, so it will not affect the enableness of the response events (P1). Secondly, there will be an enabled response to satisfy the deadline's guard, and enable the progress of time (P2).

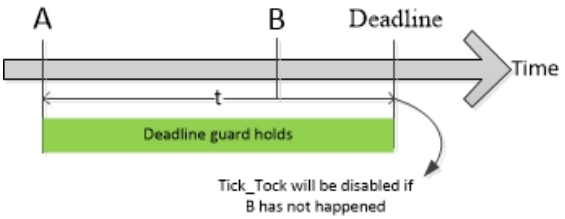


Figure 6.2: Deadline enableness diagram.

In the case of a delay, the response event will be guarded, but the *Tick_Tock* event is unguarded (P2). So, eventually the delay guard will be satisfied by the progress of time and the response event will be enabled (P1).

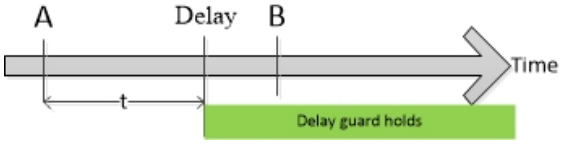


Figure 6.3: Delay enableness diagram.

Similar to a delay, an expiry will just guard the response event (P2), and even if its duration is zero, the response event will have the chance to happen at the same time as the trigger event (P1).

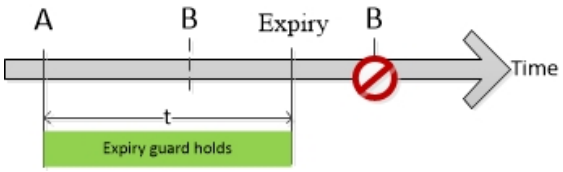


Figure 6.4: Expiry enableness diagram.

The combination of a delay and an expiry can violate P1, and combinations of a deadline with either of delay or expiry can violate P2. In the following we will go through these two cases.

The semantics of timing properties, introduced in Section 4.2, does not support having timing properties on a response event, which are triggered by alternative events. As mentioned, some of the invariants in the semantics of delay, deadline, and expiry express the order between the trigger event and its responses. Based on them if the trigger event has not happened, then none of its responses have happened neither. If we have alternative triggers, then this property will not hold anymore, and without it the POs of other invariants of timing properties' semantics cannot be discharged (e.g. check the proof of $inv2$ for event A in Section 4.2.1).

Besides there is no condition on the guards of delay and expiry, to check whether the trigger event has happened or not. It is assumed that the trigger always happens before its response. As it will be explained in Chapter 8, in parametrized timing properties, the guards of delays and expiries are conditioned, since a timing property may just include some of the possible parameters of its trigger event.

In general, the timing properties of a response event may have different trigger events, but they should not be alternative, and the response event should only be enabled, if all of its timing properties' trigger events have already happened. If there are alternative

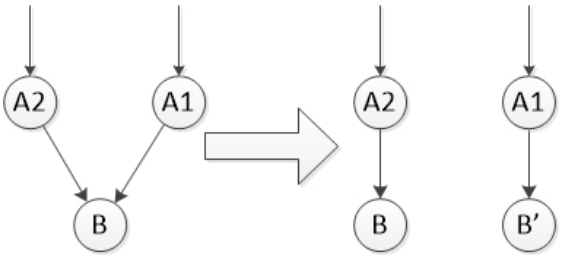


Figure 6.5: How to model alternative trigger events in Event-B (event traces diagram).

trigger events in a system, it can be modelled by duplicating the response event for each alternative trigger event, as shown in Figure 6.5.

If the timing properties of a response event are triggered by different events, it should be possible to specify the relation of their occurrence times, based on the timing properties of the system. Otherwise, the enableness preservation cannot be checked, since the duration in which a timing property will be satisfied, can be specified based on the occurrence-time of its trigger event. As a result, if the relation between the occurrence-times of the trigger events of two timing properties cannot be specified, the duration in which both of them are satisfied cannot be specified.

Assume a model with the following timing properties on its events:

$$Delay(A, C, t_1)\,, \tag{6.5a}$$

$$Expiry(B, C, t_2)\,. \tag{6.5b}$$

Based on delay (6.5a), event C is enabled if at least t_1 time-units have passed from the occurrence of A. Whereas based on expiry (6.5b), event C will not be enabled if at least t_2 time-units have passed from the occurrence of B.

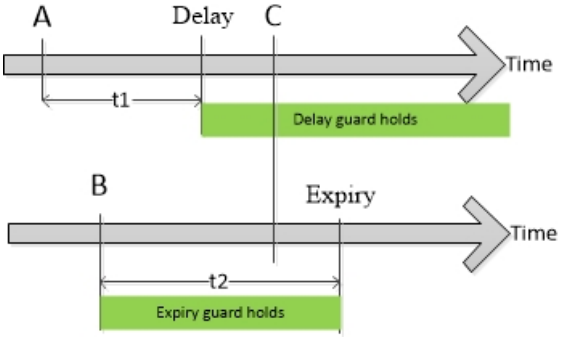


Figure 6.6: A response event which is constrained by timing properties based on different trigger events.

As shown in Figure 6.6, if we cannot specify the relation between occurrence times of A and B, it will not be possible to determine if the enableness durations of (6.5a) and

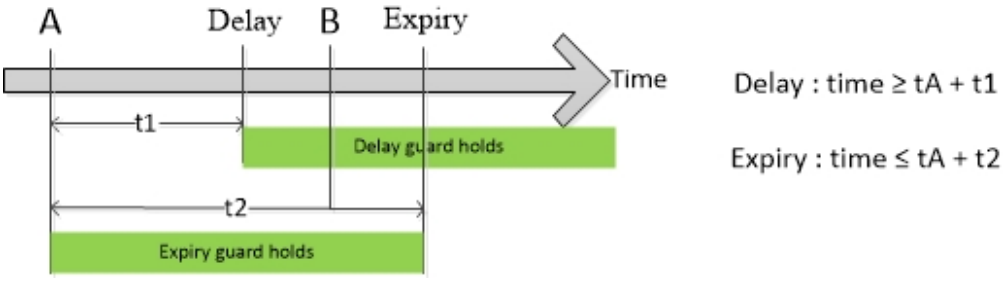
(6.5b) overlap or not. So it will not be possible to check whether event C will have the chance to happen or not.

To simplify the problem, each combination of timing properties, investigated in this chapter, will be based on the same trigger event.

6.2 Timing Properties Combination to Disable an Event indefinitely

If a response event is constrained by a delay and an expiry, a guard will be added to it, for each of those properties. If the enableness duration of the delay does not have any overlap with the enableness duration of the expiry, the response event will have no chance of occurrence.

Imagine a trigger event A and a response event B, constrained by a delay of t_1 duration and an expiry of t_2 duration. The enableness duration of B based on these two timing properties will be as follows:



To verify whether response event B will have the chance to happen based on these timing properties, the model should satisfy the following predicate:

$$\exists\ t\ \cdot\ tA + t_1\ \ \leq\ t\ \leq tA + t_2 \tag{6.6}$$

Predicate (6.6) checks whether, there is an overlap between the enableness duration of two timing properties. In this predicate tA is the integer variable which records the occurrence time of event A (trigger). This predicate can be simplified as follows:

$$t_1\ \ \leq\ \ t_2 \tag{6.7}$$

Based on (6.7) in the case of a delay and an expiry on a response event, from the same trigger event, the delay period should be less than or equal to the expiry duration.

In our case-studies, these types of properties have been expressed in terms of axioms in the context, because they specify the properties of timing properties' duration, which are all constants.

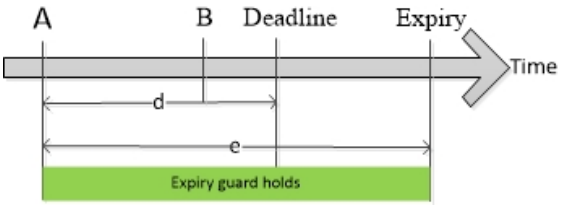
6.3 Time Progress Enableness

A badly specified deadline can disable the *Tick_Tock* event forever. To prevent this, by the end of each deadline, at least one of its response events should be enabled, if none of them has already happened.

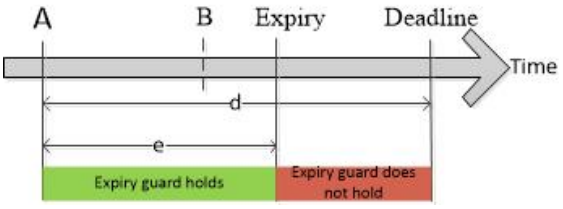
In this section, some approaches will be introduced to detect whether a deadline will be satisfied or not. Our work in time progress enableness has been inspired by deadlock freedom proof obligation by Abrial [13] which has been explained in Section 2.8.4.4.

6.3.1 A Deadline and an Expiry on a Response Event

The only sensible scenario where, a response event is constrained by a deadline and an expiry is when that deadline is based on several response events. Otherwise, either the deadline duration (d) should be within the expiry duration (e), which makes the expiry useless since the response event always happens before it has been expired,



or the expiry duration ends before the deadline duration ($e < d$),



$$A \ \wedge \ \neg \ B \ \wedge \ time \ > \ tA + e \ \Rightarrow \ \neg \ gd(B) \tag{6.8}$$

where the expiry can be passed before the occurrence of response event B, and disables it forever ($gd(X)$ represents the guards of event X).

$$A \ \wedge \ \neg \ B \ \wedge \ time \ = \ tA + d \ \Rightarrow \ \neg gd(Tick\_Tock) \tag{6.9}$$

As shown in (6.9), since the only response event has been disabled by its expiry (shown in (6.8)), when the current time gets to the end of the deadline duration, the *Tick_Tock* event will be disabled forever.

A practical combination of expiry and deadline, is when the deadline has several alternative responses, and some of them will be expired after sometimes. So some possible responses will be eliminated by the progress of time. In this case, the expiries have shorter durations than the deadline.

6.3.2 A Delay and a Deadline on a Response Event

If a response event B has been constrained by a deadline (t_1), and a delay (t_2), from an occurrence of a trigger event A, and the deadline does not have any alternative responses, the $Tick\_Tock$ event and B will be constrained as follows:

$$A \ \wedge \ \neg \ B \ \wedge \ time \ = \ tA + t_1 \Rightarrow \ \neg \ gd(Tick\_Tock)\,, \tag{6.10a}$$

$$A \ \wedge \ time \ < tA + t_2 \Rightarrow \ \neg \ gd(B)\,. \tag{6.10b}$$

Based on (6.10a) if B has not happened and the deadline duration has ended, then the $Tick\_Tock$ event is disabled. On the other hand, because of constraint (6.10b), if the delay duration has not passed yet, then B is not enabled.

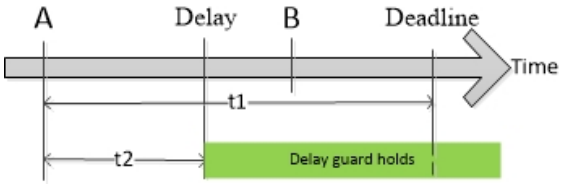


Figure 6.7: Combination of a deadline and a delay.

For event B to happen, the $Tick\_Tock$ event has to be enabled in order to progress the time, and eventually satisfy its delay:

$$A \ \wedge \ time \ < tA + t_2 \ \Rightarrow time \ < \ tA + t_1\,. \tag{6.11}$$

Predicate (6.11) can be rewritten as follow:

$$A \ \Rightarrow \ tA + t_2 \ \leq \ time \ \vee \ time < \ tA + t_1 \tag{6.12}$$

Which follows from:

$$t_2 \leq t_1 \tag{6.13}$$

As a result, the delay duration has to end before the deadline. Even, if the deadline has alternative responses, having a delay longer than a deadline on a response event, makes that response useless in the deadline, since it will be disabled throughout the deadline duration.

6.3.3 Deadline Deadlock Freedom

A generic deadline $Deadline(A, B_1 \vee .. \vee B_n, t)$, prevents the progress of time, if none of its response events have happened by the end of the deadline duration,

$$A \ \wedge \ \neg \ B_1 \ \wedge \ .. \ \wedge \ \neg \ B_n \ \wedge \ time \ = \ tA + t \Rightarrow \ \neg \ gd(Tick\_Tock). \quad (6.14)$$

For the $Tick\_Tock$ event to be eventually enabled again, at least one of the response events has to be enabled. So we need to be sure that the other timing properties of the response events do not affect this property (delay and expiry). Based on what has been explained in Sections 6.3.1 and 6.3.2 the following properties can be concluded:

DF1. The response event of a deadline which does not have alternative responses, should not be constrained by an expiry,

DF2. The response events of a deadline should not be constrained by delays, longer than the deadline.

The satisfaction of DF1 for a deadline between a trigger event A and a response event B, can be checked by a deadlock freedom proof obligation as follow:

$$time \ \leq \ tA + t \ \wedge \ A \wedge \neg B \Rightarrow gd(B) \quad (6.15)$$

Discharging proof obligation 6.15 for the $Tick\_Tock$ event, does not guarantee that the only response event of a deadline has not constrained by an expiry, but it will show that the deadline is satisfiable, and the timing progress will not be indefinitely disabled by it.

DF2 can be forced by having axiom (6.13), for every combination of a delay and a deadline. So, if the durations of all the combined delays and deadlines, satisfy (6.13), then the model holds DF2.

In general, whether a generic deadline, such as the one introduced in the beginning of the section, disables the progress of time indefinitely or not, can be checked by the following deadlock freedom proof obligation:

$$time \ \leq \ tA + t \ \wedge \ A \wedge \neg \ B_1 \wedge .. \wedge \neg \ B_n \ \Rightarrow gd(B_1) \ \vee .. \ \vee \ gd(B_n) \quad (6.16)$$

But proof obligation 6.16 can be very complex and hard to be discharged. So, a more practical approach to tackle the enableness issue, is to enforce the introduced properties, which guarantee the eventual progress of time in the model.

In the following we will go through an example to demonstrate how PO (6.16) can be used. Assume a trigger event A and its response event B as follows:

TIMING
$Delay(A, B, t_1)$
$Deadline(A, B, t_2)$
Where $t_1 > t_2$
EVENT $A \widehat{=}$
WHERE
$fA = FALSE$
THEN
$fA := TRUE$
END
EVENT $B \widehat{=}$
WHERE
$fA = TRUE$
$fB = FALSE$
THEN
$fB := TRUE$
END

(a) Events A and B plus a deadline and a delay.

INVARIANTS
(inv_1) $\neg fA \Rightarrow \neg fB$
(inv_2) $fB \Rightarrow tB \geq tA + t_1$
(inv_3) $fA \wedge \neg fB \Rightarrow time \leq tA + t_2$
(inv_4) $fB \Rightarrow tB \leq tA + t_2$

EVENT $A \widehat{=}$
WHERE
$fA = FALSE$
THEN
$fA := TRUE$
$tA := time$
END

EVENT $B \widehat{=}$
WHERE
$fA = TRUE$
$fB = FALSE$
$time \geq tA + t_1$
THEN
$fB := TRUE$
$tB := time$
END

EVENT $Tick\_Tock \widehat{=}$
any
$tick$
WHERE
$tick > 0$
$fA = TRUE \wedge$
$fB =$ FALSE$\Rightarrow$
$time + tick \leq tA + t_2$
THEN
$time := time + tick$
END

(b) Encoded delay and deadline for events A and B.

Figure 6.8: Trigger event A and its response event B, plus their timing properties

Since the delay duration is longer than the deadline of B ($t_1 > t_2$), this combination of timing properties will cause a deadlock. So we should not be able to discharge PO (6.16) for this model.

Axioms and theorems
Invariants and theorems
$\vdash$
$time \leq tA + t_2 \wedge fA =$ TRUE $\wedge$
$fB =$ FALSE $\Rightarrow gd(B)$

(Deduction)
$time \leq tA + t_2$
$fA =$ TRUE
$fB =$ FALSE
$t_1 > t_2$
$\vdash$
$fA = TRUE \wedge fB = FALSE \wedge$
$time \geq tA + t_1$

(Deduction)
$time \leq tA + t_2$
$t_1 > t_2$
$\vdash$
$time \geq tA + t_1$

$\bot$

As shown in the above proof, since the PO (6.16) cannot be discharged for this model, it is guaranteed that this combination of timing properties has caused a deadlock in our example.

6.4 Strengthening Timing Properties

In the refinement patterns, introduced in Section 4.3, the abstract timing properties are replaced by the concrete ones. But this is not always the case, sometimes in a

refinement, we are just adding some new timing properties in order to strengthen the timing properties of a model.

These new timing properties may describe the properties of a new behaviour, added in the refinement, or they may be hidden in the abstraction. Based on our experience, delays are usually hidden in the abstraction and they will be added to a model, when it includes the detailed behaviour of the corresponding system.

The same principle applies to the expiries, which strengthen the control flow of some alternative responses. By adding this type of expiries, we do not want to prove a timing refinement's consistency, such as asymmetric alternatives, explained in Section 4.3.5, but we are presenting a more concrete control flow of the events.

These new timing properties, added to a model, may affect the enableness of the events, because of the possible conflicts they may have with the existing abstract timing properties. As a result, the approaches explained in this chapter have to be applied on them, in order to guarantee the satisfaction of P1 and P2.

In the following chapter, how the approaches, we have explained so far, is used to add timing specification to an automatic gear controller case study, will be discussed.

Chapter 7

Modelling a Gear Controller

In order to check the practicality of our approach, an automatic gear controller system with several timing properties has been chosen to be model, based on what has been discussed in Chapter 4. The gear controller case study has been chosen from the UPPAAL official website [10]. The goal of modelling and verifying this case study, is to investigate the convenience of the Event-B refinement feature, in terms of modelling a real-time system. This experience helped us to enrich our refinement patterns and the semantics of deadline, delay, and expiry.

In the following sections, first the system requirements of the case study will be outlined, then our approach to model the system in Event-B, will be discussed in details. The complete Event-B model of the case study is available in Appendix A.1. In the end, how it has been modelled in UPPAAL will be explained briefly.

7.1 Gear Controller Specification

The system specification, presented in this section, is mostly based on the Lindahl etc. paper [87] on modelling and analysing a gear controller in UPPAAL. So, it will not be referenced further.

Some parts of the system specification, have been generalized based on our expectations of the case study. For example, instead of having specific values for the durations of timing properties, we just have some constants, representing the durations, and some axioms, defining their relations. In this way, the model is more generic, in a sense that it covers all the possible values of the timing properties' durations, which satisfy those axioms.

The gear controller interacts with four components of a car: the gearbox, the clutch, the engine, and the user-interface. The gear controller is responsible for synchronization of

these components, in order to set or release a gear, without causing any damage. The interactions between these components and the gear controller, happen through the car communication network, which is assumed fault free. This communication process has been modelled in terms of synchronous *Channels* in the UPPAAL model. But in our model, they will be asynchronous, since realistically, these components are connected through a communication system, which has its own delays (e.g. transfer delay).

A brief explanation of each component's functionality, will be given in the following paragraphs.

Interface
The interface is responsible for receiving the user's requests. Its user can be a driver or any component which implements the gear changing algorithm. The interface can only receive a new service request, when the previous requested service has been delivered successfully.

Gearbox
The considered gearbox is an electric one, with an electric controller. It sets a gear in 100 to 300 ms, and releases the currently engaged gear in 100 to 200 ms. If either of these two processes, takes more than the specified time, the gearbox will stop its operation, and will go to an unrecoverable error state.

Clutch
In this case-study, we have an electrically controlled clutch. It can be opened/closed in 100 to 150 ms, and if either of these two processes, takes more than 150 ms, the clutch will go to an unrecoverable error state.

Engine
The engine has the following operating modes:

- **Zero Torque Difference** mode is when the engine, and the gearbox have the same torque,
- **Synchronous Speed** mode is when the engine, and the gearbox have the same speed,
- **Normal Torque** mode is the normal mode, when the engine has the specified torque by the driver. This mode exists in the UPPAAL model, but since its accomplishment, is not part of the gear changing process (not required), and it is part of the detailed behaviour of the engine, it has been excluded (abstracted), in our Event-B model.

As for the other components, there are some time properties on the accomplishment of the engine's operation modes. The maximum acceptable latency to accomplish a zero

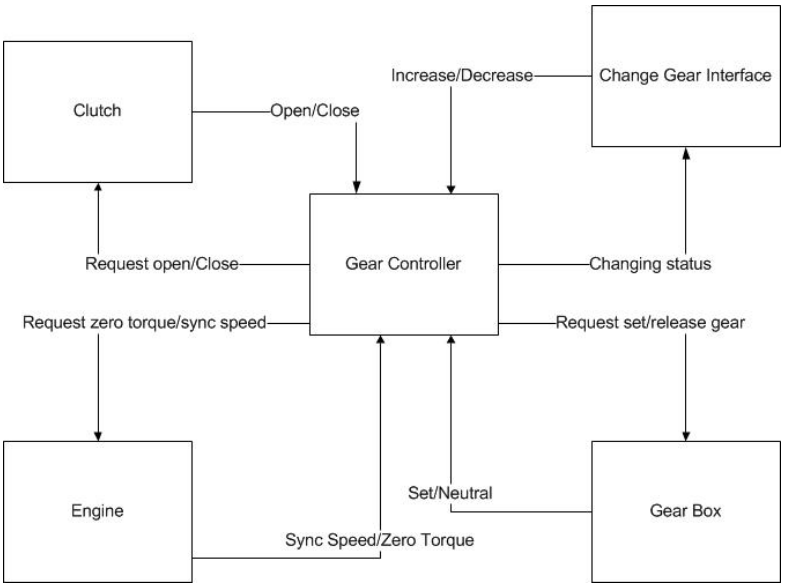


Figure 7.1: Interactions Between Gear Controller Components

torque difference is 400ms, and this duration is 200ms for the speed synchronization process, otherwise the process will be aborted.

Gear Controller

When the gear controller receives a request from the interface, it tries to accomplish the service (changing the gear), in four steps. Those steps are as follows:

1. Gaining the zero torque difference between the engine, and the gearbox,
2. Releasing the currently engaged gear,
3. Gaining the synchronous speed between the engine, and the gearbox,
4. Setting the requested gear.

After setting the requested gear, system is ready to receive the next request. These steps are required to accomplish a gear-change, in a normal situation. In difficult situations, the engine may not gain the zero-torque difference, or the synchronous speed, in time. This internal fault of the system can be overcome by opening the clutch. Opening the clutch disengages the engine from the gearbox, and makes it possible to release the current gear or set the requested gear, without gaining the zero torque or the synchronous speed. In this case closing the clutch will safely bridge the speed, and the torque difference, between the gearbox and the engine. So, there are some levels of fault tolerance in this system.

Figure 7.1 shows how the concerned components, interact with each others during the gear-changing process.

This section aimed to explain the functionality of the four effective components, in the process of changing the currently engaged gear, to the requested one. In the following, the requirements of the gear controller system, will be presented.

7.1.1 System Requirements

In this section, the requirements of the gear controller case-study, will be discussed in five categorizes, performance requirements, functional requirements, error detection requirements, and the environment assumptions. Since, timing properties play an important role in synchronization, and error detection in this system, many of the requirements focus on them.

7.1.1.1 Performance

These requirements, specify the tolerable latencies of the system's responses.

P1. The gear change request should be responded within 1.5 second either by accomplishment of the requested gear or occurrence of an unrecoverable error,

P2. The controller should be deadlock free,

P3. The controller has to ask for an opened clutch, if the zero torque difference has not been gained by the engine, within 255 ms,

P4. The controller has to ask the clutch to be opened, if the speed synchronization has not been provided by the engine within 155 ms.

7.1.1.2 Functionality

The following requirements aim to specify the desirable functionality of the gear controller system.

F1. It should be possible to set any requested gear, unless an unrecoverable error has happened,

F2. There are three gear changing scenarios:

Changing from the neutral gear,

Changing to the neutral gear,

Changing an engaged gear to another,

F3. To change the gear, first the currently engaged gear should be released, unless the gear is neutral,

F4. To change the gear, the requested gear should be set after releasing the current gear, unless the neutral gear is requested,

F5. To release the currently engaged gear, a zero torque difference between the engine and the gear box should be gained,

F6. To set the requested gear, the speed of the gearbox and the engine should be synchronized,

F7. Clutch may be used to set or release gears,

F8. By Opening the clutch, there is no need to have a zero torque difference between the engine and the gearbox, to release the currently engaged gear,

F9. By Opening the clutch, there is no need to have a synchronous speed between the engine and the gearbox, in order to set the requested gear,

7.1.1.3 Error Detection

These requirements specify the unrecoverable error states, which may happen in the controller during the gear changing process.

E1. If opening the clutch has not been accomplished within 200ms,

E2. If releasing the current gear has not been accomplished within 250ms,

E3. If setting the requested gear has not been accomplished within 350ms,

E4. If closing the clutch has not been accomplished within 200ms.

7.1.1.4 Environment Assumptions

These assumptions specify the behaviour of the gear controller's environment and its properties. These assumptions are critical in developing the right controller.

The Engine Assumptions:

EA1. It should be possible to use the engine, to gain the synchronous speed between the engine and the gearbox,

EA2. It should be possible to use the engine, to gain the zero torque difference between the engine and the gearbox,

The Gearbox Assumptions:

EA3. It should be possible to use the gearbox, in order to set or release a gear, unless an unrecoverable error has happened in the gearbox,

EA4. When gearbox has not been successful to set the gear within 300ms, it goes to an unrecoverable error state,

EA5. When gearbox has not been successful to release the gear within 200ms, it goes to an unrecoverable error state,

The Clutch Assumptions:

EA6. It should be possible to open or close the clutch, unless the clutch reaches an unrecoverable error state,

EA7. When clutch has not been opened within 150ms, it goes to an unrecoverable error state,

EA8. When clutch has not been closed within 150ms, it goes to an unrecoverable error state.

In this case study, to generalize the model, symbolic values have been used for the timing properties' durations. As the result, the timing properties' durations are constants which their relations have been specified in terms of axioms. Table 7.1 presents the constant, we have used in our model.

Constant	Description	Component
ChangeDL	Deadline duration for responding a gear change request	Controller
S_FN	Deadline duration for accomplishing the setting step (changing from the neutral)	Controller
R_NN	Deadline duration for accomplishing the releasing step (changing from a gear to another)	Controller
S_NN	Deadline duration for accomplishing the setting step (changing from a gear to another)	Controller
S_NN_RC	Deadline duration for accomplishing the setting step when the clutch is open (changing from a gear to another)	Controller
R_TN	Deadline duration for accomplishing the releasing step (changing to the neutral)	Controller

R_NN_EX	Expiry duration for accomplishing releasing step, without penning the clutch (changing from a gear to another)	Controller
SyncOpen_DL	Deadline duration for gaining the sync speed or opening the clutch	Controller
Sync_EX	Expiry duration for gaining the sync speed	Controller
SetGear_DL	Deadline duration for setting the requested gear or an error occurrence	Controller
CloseClutch_DL	Deadline duration for closing the clutch or an error occurrence	Controller
ZeroOpen_DL	Deadline duration for gaining the zero-torque or opening the clutch	Controller
Zero_EX	Expiry duration for gaining the zero-torque	Controller
Release_DL	Deadline duration for releasing the current gear or an error occurrence	Controller
OpenClutch_Sync_DE	Delay duration of asking for an open clutch, instead of the sync speed	Controller
OpenClutch_Zero_DE	Delay duration for asking an open clutch instead of the zero-torque	Controller
Sync_DL	Deadline duration for synchronizing the speed or asking for an open clutch	Controller
Zero_DL	Deadline duration for gaining the zero-torque difference or asking for an open clutch	Controller
OpenClutch_DL	Deadline duration for the opening the clutch or an error occurrence	Controller
Channel_DL	Deadline duration of transferring messages through the communication system	Channel
Enigne_Sync_DL	Deadline duration for synchronizing the speed or giving up	Engine
Enigne_Zero_DL	Deadline duration for gaining zero-torque difference or giving up	Engine
Clutch_Open_DL	Deadline duration for opening the clutch or an error occurrence	Clutch
Clutch_Close_DL	Deadline duration for closing the clutch or an error occurrence	Clutch

Gear_Set_DL	Deadline duration for setting a gear or an error occurrence	Gearbox
Gear_Release_DL	Deadline duration for releasing a gear or an error occurrence	Gearbox

Table 7.1: The constants, used as the timing properties' durations of the gear controller case-study.

In this section, the system requirements of the gear-controller system have been explained.

7.1.2 Refinement Strategy

So far the requirements of the case-study have been introduced. In this section our strategy of including those requirements in the model, step by step, by each refinement will be explained.

In the most abstract level, $P1$, $F1$ have been included, by having three events, representing the request, successful change, and error occurrence, plus their timing property. The first refinement adds $F2$ to the model by adding the three possible changing scenarios. The second refinement introduced the use of clutch which covers $F7$. The third refinement adds $F3$, $F4$, and $F7$ by introducing the required steps to respond a change for each scenario ($F2$). In the fourth refinements the timing properties of the required steps, replaces the abstract ones. The fifth, the sixth, the seventh, and the eighth refinements add $P3$, $P4$, $F5$, $F6$, $F8$, $F9$, and the error detection requirements explained in Section 7.1.1.3 to the model, for all the introduced changing scenarios in $F2$. By the ninth refinement, all the environment assumptions introduced in Section 7.1.1.4 have been included in the model by adding the clutch, gear, and the engine events to the model. As it will be explained in Section 7.2.11, the last refinement has been dedicated to provide the required changes, before the decomposition.

The deadlock freeness (exP2) has been checked in the untimed model, by using the model checker in Rodin tool-set, and then in the timed model based on what has been explained in Chapter 6, some axioms have been declared on timing properties duration, which guarantee the satisfaction of enableness properties P1 and P1.

In the following we will go through the stepwise process of modelling, and verifying the gear-controller system in Event-B.

7.2 Event-B Model of The Gear Controller

The first thing to do in order to model the gear-controller system, is to decide on the scope of the most abstract machine. Then in each refinement we have to decide on the details, we want to add to the model. The timed gear controller system has been modelled in 10 levels of abstraction in Event-B.

In the following, the stepwise process of modelling the gear controller system and its timing properties, in Event-B, based on the approaches introduced in Chapter 4, will be explained.

7.2.1 The Most Abstract Machine and Context

In the most abstract machine, the main functionality of the gear controller, which is responding to a gear change request, has been modelled by three events. The *Request* event represents the occurrence of a gear change request, the *Response* event represents the successful accomplishment of a change, and the *Error* event represents the occurrence of an unrecoverable error, during the response process.

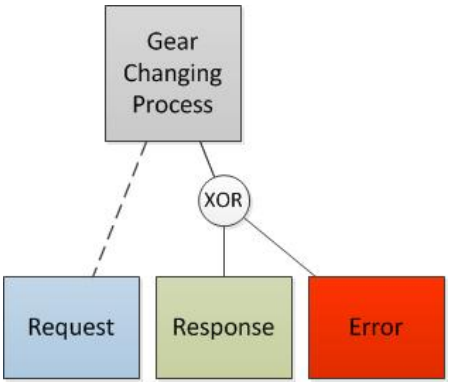


Figure 7.2: The most abstract machine

As shown in Figure 7.2, there is a non-deterministic choice between the successful and unsuccessful responses. In the timed model, there is a deadline on successful and unsuccessful events, from the occurrence of the request event, as follows:

$$Deadline(Request,\ Response \lor Error,\ ChangingDL) \tag{7.1}$$

Where *ChangingDL* is a constant, representing the possible durations of the deadline. This timing property specifies the maximum acceptable latency to respond to a gear change request.

In order to give an idea of a typical Event-B machine, and how the most abstract model has been constructed in this study, the most abstract machine and context of the gear controller system, is presented in the following:

An Event-B Specification of c0
Creation Date: 12Jul2012 @ 00:45:52 PM

CONTEXT c0
CONSTANTS

$ChangeDL$
AXIOMS

axm1 : $ChangeDL > 0$
END

An Event-B Specification of m0
Creation Date: 5Jul2012 @ 04:33:34 PM

MACHINE m0
VARIABLES

$Request$

$Response$

$Error$
INVARIANTS

inv1 : $Request \in BOOL$

inv2 : $Error \in BOOL$

inv3 : $Response \in BOOL$
TIMING

tim1 : $Deadline(Request, Response \lor Error, ChangingDL)$
EVENTS
Initialisation
begin

act1 : $Error := FALSE$

act2 : $Request := FALSE$

act3 : $Response := FALSE$
end
Event $Request \widehat{=}$
when

grd1 : $Request = FALSE$

then

act1 : $Request := TRUE$

end

Event *Response* $\widehat{=}$

when

grd1 : $Request = TRUE$
grd2 : $Error = FALSE$
grd3 : $Response = FALSE$

then

act1 : $Response := TRUE$

end

Event *Error* $\widehat{=}$

when

grd1 : $Error = FALSE$
grd2 : $Request = TRUE$
grd3 : $Response = FALSE$

then

act1 : $Error := TRUE$

end

Event *FINAL* $\widehat{=}$

when

grd1 : $Response = TRUE$

then

act1 : $Request := FALSE$
act2 : $Response := FALSE$

end

END

Axiom *axm1* in the context, which expresses a deadline duration greater than zero, is not required for the proofs. It has been added, because it is a property of the system.

As shown, beside the **INITIALISATION** event, which specifies the initial values of the system state variables, and the *FINAL* event, which resets the occurrence flags, if a request has been responded successfully, there are 3 other events, which model the most abstract transitions, between different states of the gear controller, in order to gain the requested gear.

The main reason, to have a separate event to reset the flags and not to do it in the response event (as the last event of each iteration), is the deadline semantics. As explained in Section 4.2.3, based on deadline semantics, the response event has to have

an occurrence flag to define the guard of the *Tick_Tock* event. Since it is not possible to reset the flags, in an event which itself has a flag, we need to have a separate event to do it. Besides, as Salehi's work [61] shows, by resetting the occurrence flags in the last event of an iteration, if later on that event is refined by several alternative events, the resetting actions has to be repeated in each of them, which can cause redundancy. As a result, we suggest a separate event for this mean. But, this event has not been considered as a part of timing properties' semantics, because it is an order related event.

7.2.2 The Second Level of Abstraction

In this level of abstraction, the three possible scenarios of changing a gear have been introduced. Consequently the abstract deadline has been refined by three alternative concrete deadlines, based on the approach explained in Section 4.3.4 (refining an abstract deadline by alternative concrete deadlines).

These three scenarios are based on whether the currently engaged gear, or the requested gear, are neutral. As mentioned before, in order to change the gear, first the currently engaged gear should be released, and then the requested one has to be engaged. If no gear is engaged currently (neutral), the releasing step will be irrelevant. If the neutral gear is requested, then just by releasing the currently engaged gear, the request has been satisfied. As show in Figure 7.3, the three possible gear-changing scenarios are as

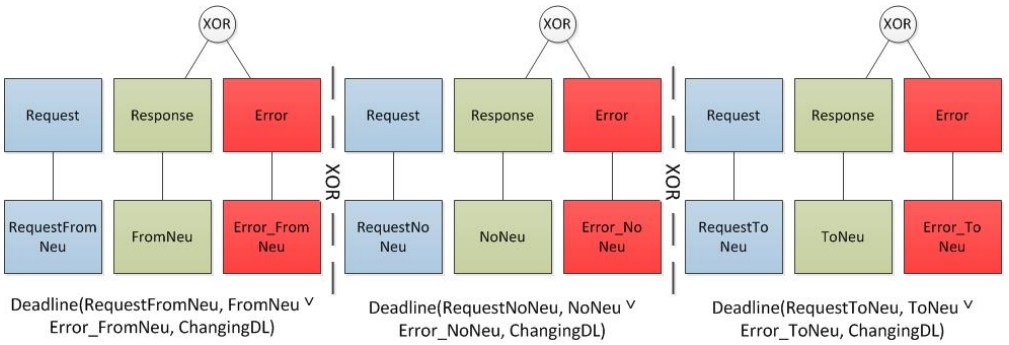


Figure 7.3: Representing the relation of the concrete and abstract events, based on the first refinement.

follows:

- *FromNeu*: Changing from the neutral gear to an engaged gear,
- *NoNeu*: Changing the currently engaged gear to another,
- *ToNeu*: Changing the currently engaged gear to neutral.

As shown, the abstract sequence of request-response has been refined by three alternative sequences. As a result, the concrete deadlines will be as follows:

$$Deadline(RequestFromNeu,\ FromNeu\ \vee\ Error\_FromNeu, ChangingDL)\,, \tag{7.2a}$$

$$Deadline(RequestNoNeu,\ NoNeu\ \vee\ Error\_NoNeu,\ ChangingDL)\,, \tag{7.2b}$$

$$Deadline(RequestToNeu,\ ToNeu\ \vee Error\_ToNeu,\ ChangingDL)\,. \tag{7.2c}$$

Based on which sequence is activated by the occurrence of its request event, the corresponding concrete deadline will satisfy the abstract deadline (7.1).

7.2.3 The Third Level of Abstraction

As mentioned before the gear-changing process can be done with or without opening the clutch. In this refinement, these two possible cases have been added to the model. So each event, representing the successful accomplishment of a change, will be refined by two alternative cases, with opening the clutch or without it. Accordingly, the abstract deadlines will be refined based on the approach explained in Section 4.3.3 (case split of response).

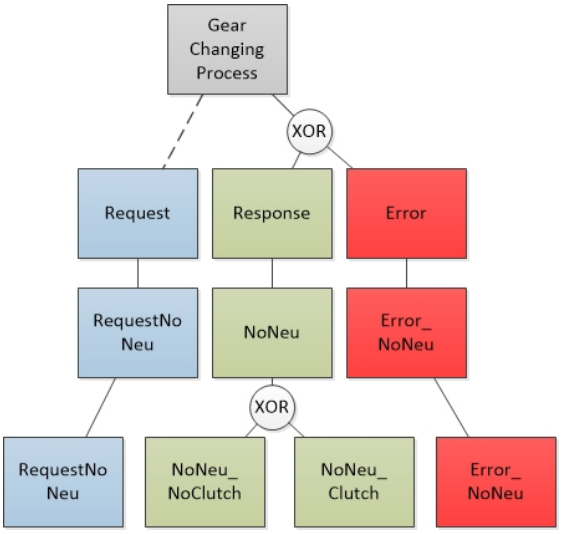


Figure 7.4: Adding the clutch use, in order to change the engaged gear to another.

The refinement diagram of changing the engaged gear to another (*NoNeu* scenario), presented in Figure 7.4. The *NoNeu_NoClutch* event represents a successful change without using the clutch, and event *NoNeu_Clutch* represents the case where the clutch has been opened, in order to change a gear successfully. The same refinement pattern

was used for the refinement of the other scenarios. Based on the introduced cases, the concrete deadlines will be as follows:

$$Deadline(RequestFromNeu,\ FromNeu\_NoClutch \vee FromNeu\_Clutch \vee Error\_FromNeu,\ ChangingDL)\ replaces\ (7.2a)\,, \tag{7.3a}$$

$$Deadline(RequestNoNeu,\ NoNeu\_NoClutch \vee NoNeu\_Clutch \vee Error\_NoNeu,\ ChangingDL)\ replaces\ (7.2b)\,, \tag{7.3b}$$

$$Deadline(RequestToNeu,\ ToNeu\_NoClutch \vee ToNeu\_Clutch \vee Error\_ToNeu,\ ChangingDL)\ replaces\ (7.2c)\,. \tag{7.3c}$$

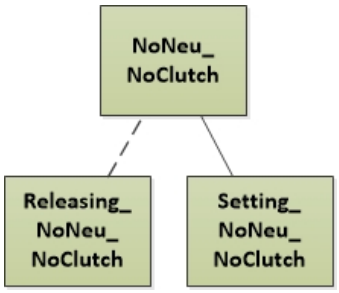
As shown above, based on the introduced refinement approach in Section 4.3.3, disjunction of alternative concrete responses, have replaced the abstract successful response, in each concrete deadline.

7.2.4 The Fourth Level of Abstraction

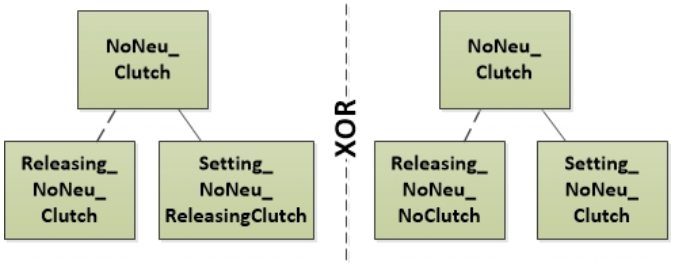
In the fourth level of abstraction, the sequence of releasing the current gear and setting the requested gear replaces the abstract successful gear-change. As mentioned before, when the gear is currently neutral (*FromNeu_NoClutch* or *FromNeu_Clutch*), the releasing step is irrelevant, and when the neutral gear is requested (*ToNeu_NoClutch* or *ToNeu_Clutch*), the releasing step is sufficient.

Changing from an engaged gear to another gear is the most complex scenario, since both of the concrete steps are required and using the clutch in the first step, affects the following one. As shown in Figure 7.5, if the releasing step has been accomplished without opening the clutch (*NoNeu_Releasing_NoClutch*), the setting step can be done with or without opening the clutch. Whereas, if the clutch has been opened to release the engaged gear, it will remain open during the setting process. Besides, the occurrence of an unrecoverable error has been refined by two alternative cases, occurrence of an unrecoverable error during the releasing process (*Error_Releasing_NoNeu*), or during the setting process (*Error_Setting_NoNeu*).

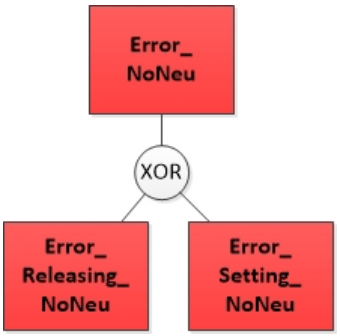
As shown in Figure 7.6, each of the abstract events *FromNeu_NoClutch* and *FromNeu_Clutch*, representing a successful change, when the gear was initially neutral, is refined by two events, representing the setting process with or without using the clutch (*FromNeu_Setting_NoClutch* or *FromNeu_Setting_Clutch*). Accordingly, each of the abstract events *ToNeu_NoClutch* and *ToNeu_Clutch*, is refined by two events representing the releasing process with or without using the clutch, when the neutral gear has been requested (*ToNeu_Releasing_NoClutch* or *ToNeu_Releasing_Clutch*).



(a) Changing the currently engaged gear to another, without opening the clutch.



(b) Changing the currently engaged gear to another, by opening the clutch.



(c) Error refinement diagram.

Figure 7.5: Refinement Diagram: Introducing the required steps to change an engaged gear to another gear.

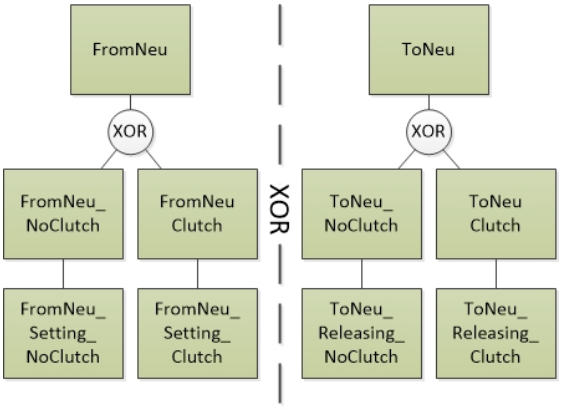


Figure 7.6: Refinement diagram of changing from/to neutral gear.

Since the control flow refinement is complex by itself, in this level of abstraction, applying the timing refinement makes the model overly complicated. As a result, replacing the abstract timing properties, by sequential concrete sub-timing properties, has been postponed until the next refinement. This feature is one of the advantages of our approach.

When an abstract event has been replaced by a sequence of concrete events, the occurrence of the last event in the concrete sequence, is equivalent to the occurrence of the

abstract event. As a result, if a response event of an abstract timing property, is refined by a sequence of concrete sub-responses, replacing it by the last event of the concrete sequence, in the concrete timing property, will satisfy the abstract one. Consequently, breaking the overall timing property into a sequence of concrete timing properties, can be done in the following refinements.

Accordingly, the abstract timing properties will be changed as follows in this refinement:

$$\begin{aligned} &Deadline(RequestFromNeu,\ FromNeu\_Setting\_NoClutch \\ &\quad \vee\ FromNeu\_Setting\_Clutch \\ &\quad \vee\ Error\_FromNeu, ChangingDL)\ replaces\ (7.3a)\,, \end{aligned} \tag{7.4a}$$

$$\begin{aligned} &Deadline(RequestNoNeu,\ NoNeu\_Setting\_NoClutch \\ &\quad \vee\ NoNeu\_Setting\_Clutch \vee NoNeu\_Setting\_ReleasingClutch \\ &\quad \vee Error\_Releasing\_NoNeu \\ &\quad \vee\ Error\_Setting\_NoNeu, ChangingDL)\ replaces\ (7.3b)\,, \end{aligned} \tag{7.4b}$$

$$\begin{aligned} &Deadline(RequestToNeu,\ ToNeu\_Releasing\_NoClutch \\ &\quad \vee ToNeu\_Releasing\_Clutch \\ &\quad \vee Error\_ToNeu, ChangingDL)\ replaces\ (7.3c)\,. \end{aligned} \tag{7.4c}$$

As shown in deadline (7.4b), instead of breaking the abstract deadline, to sequential sub-deadlines,

1. Deadline between the occurrence of the request, and the occurrence of the releasing or the occurrence of an unrecoverable error,
2. Deadline between the occurrence of the releasing, and either of the setting or an error occurrences,

The abstract successful responses (*NoNeu_NoClutch* and *NoNeu_Clutch*), have been replaced by the last sub-response events of the concrete sequences (*NoNeu_Setting_NoClutch*, *NoNeu_Setting_Clutch*, and *NoNeu_Setting_ReleasingClutch*). Besides, the error occurrence has been replaced by its alternative concrete events (*Error_Releasing_NoNeu* or *Error_Setting_NoNeu*), based on the refinement pattern, explained in Section 4.3.3.

7.2.5 The Fifth Level of Abstraction

In the fifth level of abstraction, the abstract deadline of changing the currently engaged gear to another, is replaced by its concrete sub-deadlines. What makes this refinement challenging, is the need for a concrete expiry in order to satisfy the abstract deadline.

The timing refinement has been done based on the approach explained in Section 4.3.5 (Asymmetric Alternatives). The concrete deadlines of *NoNeu* scenario (changing the currently engaged gear to another) are as follows:

From Neutral (*FromNeu* Scenario):

$$Deadline(RequestFromNeu,\ FromNeu\_Setting\_NoClutch \vee FromNeu\_Setting\_Clutch \vee Error\_FromNeu, S\_FN)\ replaces\ (7.4a)\,, \tag{7.5a}$$

Where

$$S\_FN \leq ChangingDL\,. \tag{7.5b}$$

From Gear to Another Gear (*NoNeu* Scenario):

$$Deadline(RequestNoNeu,\ NoNeu\_Releasing\_NoClutch \vee NoNeu\_Releasing\_Clutch \vee Error\_Releasing\_NoNeu, R\_NN)\ replaces\ (7.4b)\,, \tag{7.6a}$$

$$Deadline(NoNeu\_Releasing\_NoClutch,\ NoNeu\_Setting\_NoClutch \vee NoNeu\_Setting\_Clutch \vee Error\_Setting\_NoNeu, S\_NN)\ replaces\ (7.4b)\,, \tag{7.6b}$$

$$Deadline(NoNeu\_Releasing\_Clutch,\ NoNeu\_Setting\_ReleasingClutch \vee Error\_Setting\_NoNeu, S\_NN\_RC)\ replaces\ (7.4b)\,, \tag{7.6c}$$

Where

$$R\_NN + S\_NN\_RC \leq ChangingDL\,. \tag{7.6d}$$

To Neutral (*ToNeu* Scenario):

$$Deadline(RequestToNeu,\ ToNeu\_Releasing\_NoClutch \vee ToNeu\_Releasing\_Clutch \vee Error\_ToNeu, R\_TN)\ replaces\ (7.4c)\,, \tag{7.7a}$$

Where

$$R\_TN \leq ChangingDL\,. \tag{7.7b}$$

Deadlines (7.5a) and (7.7a) replace the abstract durations with the concrete ones based on the required time, for releasing or setting processes. On the other hand deadlines (7.6a), (7.6b) and (7.6c) are the sequential sub-deadlines, which replace the *NoNeu* scenario's abstract deadline. Based on how the current gear is released, there are two alternative sequences of concrete deadlines, as follows:

1. If the currently engaged gear is released, without using the clutch, then the sequence of sub-deadlines (7.6a) and (7.6b) has to satisfy the abstract deadline,

2. If the clutch has been opened during the releasing process, then the sequence of sub-deadlines (7.6a) and (7.6c) has to satisfy the abstract deadline.

In case 1, the refinement is consistent, since the sequence of the concrete deadlines does not exceed the abstract deadline ($R\_NN + S\_NN \leq ChangingDL$).

But, case 2 is more complex. In this case, we are not necessarily assuming, the satisfaction of the abstract deadline 7.4b, by the sequence of the concrete sub-deadlines (7.6a) and (7.6b). As a result, during the proof, the modeller will recognize that more properties are required to satisfy the consistency of the refinement.

Based on the system specification, the controller first tries to accomplish the neutral gear (releasing the current gear) without using the clutch. After putting some efforts, if it has not been successful, it tries to open the clutch. So, after some duration, releasing without opening the clutch will be expired.

Deadline (7.6a) specifies the maximum duration in which, either of error, or gear disengagement, with or without opening the clutch, has to happen. By having the expiry for releasing the current gear without opening the clutch, sufficient details of the system timing properties will be available to satisfy the abstract deadline. This expiry will be as follows:

$$Epiry(RequestNoNeu,\ NoNeu\_Releasing\_NoClutch,\ R\_NN\_EX)\ replaces\ (7.4b)\,, \tag{7.8a}$$

Where

$$R\_NN\_EX\ + S\_NN\ \leq ChangingDL\,. \tag{7.8b}$$

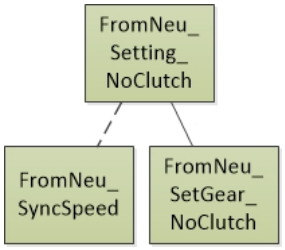
Based on expiry (7.8a) it is guaranteed that if the *NoNeu_Releasing_NoClutch* event has happened, it was within *R_NN_EX* time-units of the request, which followed by deadline (7.6c), will satisfy the abstract deadline (deadline (7.4b)). As a result, deadline 7.4b has been replaced by concrete timing properties 7.6a, 7.6b, 7.6c and 7.8a.

7.2.6 The Sixth Level of Abstraction

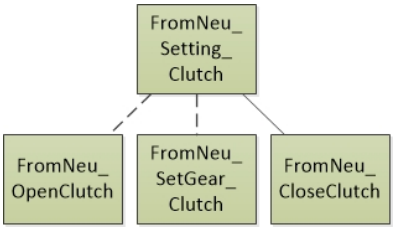
Based on the system specification, some steps are needed to be accomplished in order to release the currently engaged gear, and set the requested gear. Concerning the complexity of the refinements, the sub-steps of each scenario, have been added to the model in different refinements. The following three refinements are dedicated to add these steps.

In this level of abstraction, the steps needed to be accomplished by the gear controller to set the requested gear, when the gear is currently neutral have been added to the model. The corresponding abstract timing properties have been refined accordingly.

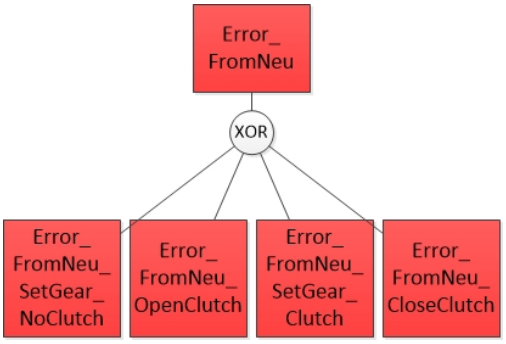
As shown in Figure 7.7, and explained in the requirements (Section 7.1.1), first the engine has to synchronize its speed with the gearbox (*FromNeu_SyncSpeed*), or the clutch has to be opened (*FromNeu_OpenClutch*). If the synchronous speed has been gained in time, either the requested gear will be engaged successfully (*FromNeu_SetGear_NoClutch*), or an unrecoverable error will happen (*Error_FromNeu_SetGear_NoClutch*). But if the clutch has been opened, and the gear has been set successfully (*FromNeu_SetGear_Clutch*), in order to accomplish the service, the clutch has to be closed (*FromNeu_CloseClutch*). So, if no error happens during the closing process (*Error_FromNeu_CloseClutch*), the requested service will be provided successfully.



(a) Introducing the steps required to set the requested gear without the clutch.



(b) Introducing the steps required to set the requested gear with the clutch.



(c) Introducing the unrecoverable errors, which may occur during setting the requested gear.

Figure 7.7: Refinement diagram of setting the requested gear, when the gear was neutral before the request.

Since the abstract successful responses have been refined by sequential concrete events, their timing properties can be refined based on the approach, explained in Section 4.3.1. The concrete deadlines of the *FromNeu* scenario (changing gear, when initially the gear was neutral), are as follows:

$$\begin{aligned} &Deadline(RequestFromNeu,\ FromNeu\_SyncSpeed \\ &\quad \vee\ FromNeu\_OpenClutch \\ &\quad \vee\ Error\_FromNeu\_OpenClutch,\ SyncOpen\_DL)\ replaces\ 7.5a\,, \end{aligned} \tag{7.9a}$$

$$\begin{aligned} &Deadline(FromNeu\_SyncSpeed,\ FromNeu\_SetGear\_NoClutch \\ &\quad \vee\ Error\_FromNeu\_SetGear\_NoClutch,\ SetGear\_DL)\ replaces\ 7.5a\,, \end{aligned} \tag{7.9b}$$

$$Deadline(FromNeu\_OpenClutch,\ FromNeu\_SetGear\_Clutch \vee Error\_FromNeu\_SetGear\_Clutch,\ SetGear\_DL)\ replaces\ 7.5a\,, \tag{7.9c}$$

$$Deadline(FromNeu\_SetGear\_Clutch,\ FromNeu\_CloseClutch \vee Error\_FromNeu\_CloseClutch, CloseClutch\_DL)\ replaces\ 7.5a\,, \tag{7.9d}$$

Where

$$SyncOpen\_DL\ +\ SetGear\_DL\ +\ CloseClutch\_DL \leq S\_FN\,. \tag{7.9e}$$

Since the sequence of the sequential sub-deadlines, satisfies the abstract deadline, the timing refinement is consistent.

Besides refining the abstract deadline, an expiry has been introduced to strengthen the timing properties of event *FromNeu_SyncSpeed*, as explained in Section 6.4. This expiry property is as follows:

$$Expiry(RequestFromNeu,\ FromNeu\_SyncSpeed,\ Sync\_EX)\,, \tag{7.10a}$$

Where

$$Sync\_EX \leq SyncOpen\_DL\,. \tag{7.10b}$$

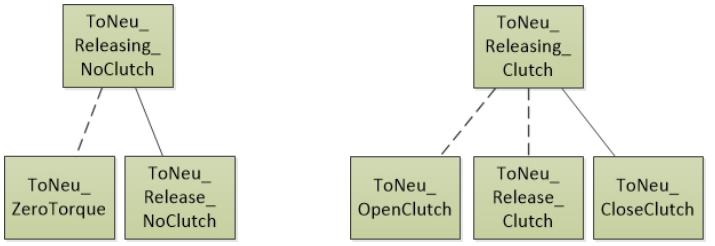
Based on expiry (7.10a), waiting for engine to gain a synchronous speed with the gearbox will be expired after sometimes and the only possible responses will be opening the clutch or an occurrence of an unrecoverable error. This timing property is not required in order to satisfy an abstract timing property. By adding it, we just present a more precise model of the control flow.

Axiom 7.10b, is not required to prove the consistency of the refinement, it is just a trivial specification of the system. As explained in Section 6.3.1, having an expiry on an event, longer than its deadline, is useless.

7.2.7 The Seventh Level of Abstraction

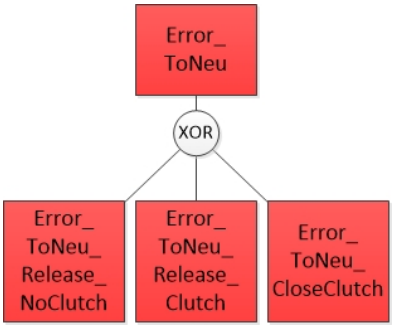
Similar to what has been explained in previous section, the required steps to release the currently engaged gear, in response to the user's neutral gear request, are added in this level of abstraction.

As shown in Figure 7.8, first the controller tries to synchronize the speeds of the engine and the gearbox (*ToNeu_ZeroTorque*), or open the clutch (*ToNeu_OpenClutch*). If the synchronous speed is gained, or the clutch has been opened successfully, either the currently engaged gear will be released before its deadline or an unrecoverable error will



(a) Introducing the steps required to release the currently engaged gear without the clutch.

(b) Introducing the steps required to release the current gear with the clutch.



(c) Introducing the unrecoverable errors, which may occur during the gear changing process.

Figure 7.8: Refinement diagram of releasing the currently engaged gear, when the neutral gear is requested.

happen. If the clutch is opened, the clutch has to be closed, in order to finalize the gear-changing process. If the clutch has not been closed by its deadline, an unrecoverable error will happen, and consequently the requested service cannot be delivered.

In the end, if the clutch has been opened, it should be closed (*ToNeu_CloseClutch*), and if no error (*Error_ToNeu_CloseClutch*) has happened during the closing process, the service is successfully delivered.

Since the successful response has been refined by sequential concrete sub-responses, the abstract deadlines has been refined based on the approach, explained in Section 4.3.1. The concrete deadlines are as follows:

$$Deadline(RequestToNeu,\ ToNeu\_ZeroTorque \vee ToNeu\_OpenClutch \vee Error\_ToNeu\_OpenClutch,\ ZeroOpen\_DL)\,, \tag{7.11a}$$

$$Deadline(ToNeu\_ZeroTorque,\ ToNeu\_Release\_NoClutch \vee Error\_ToNeu\_Release\_NoClutch, Release\_DL)\,, \tag{7.11b}$$

$$Deadline(ToNeu\_OpenClutch,\ ToNeu\_Release\_Clutch \vee Error\_ToNeu\_Release\_Clutch, Release\_DL)\,, \tag{7.11c}$$

$$Deadline(ToNeu\_Release\_Clutch,\ ToNeu\_CloseClutch \\ \vee\ Error\_ToNeu\_CloseClutch, CloseClutch\_DL)\,, \tag{7.11d}$$

Where

$$ZeroOpen\_DL\ +\ Release\_DL\ +\ CloseClutch\_DL\ \leq R\_TN\,. \tag{7.11e}$$

Similar to previous refinement, since the sequence of the *FromNeu* scenario's concrete deadlines, satisfies its abstract deadline, the refinement is consistent.

Besides, an expiry is added in this refinement, which strengthens the timing properties, on the occurrence of the *ToNeu_ZeroTorque* event (based on Section 6.4). This expiry is constructed as follows:

$$Expiry(RequestToNeu,\ ToNeu\_ZeroSpeed,\ Zero\_EX)\,, \tag{7.12a}$$

Where

$$Zero\_EX \leq ZeroOpen\_DL\,. \tag{7.12b}$$

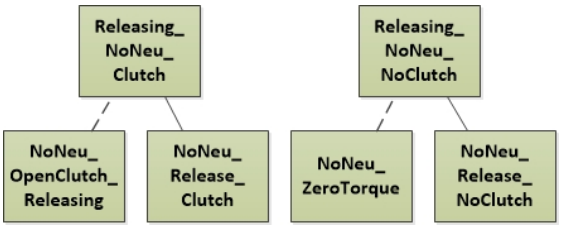
Axiom (7.12b), is not required to prove the consistency of the refinement. It is a trivial specification of the expiry duration. As explained in Section 6.3.1, having an expiry on a response event, longer than its deadline, is useless.

7.2.8 The Eighth Level of Abstraction

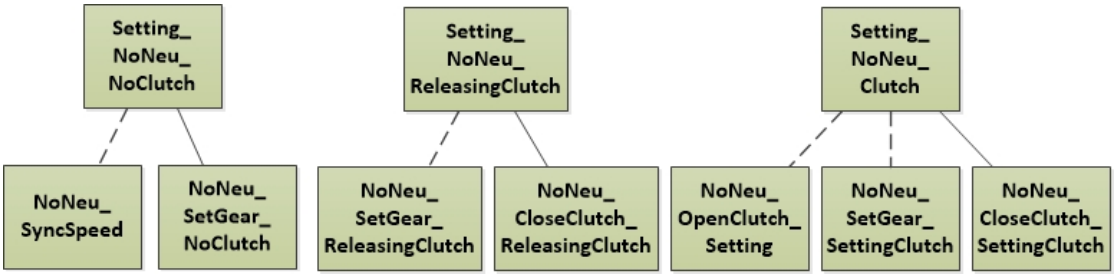
In this level of abstraction, the required steps to release the currently engaged gear and set the requested gear have been added to the model. These steps are similar to what have been explained, in two previous refinements. In the *NoNeu* scenario (changing the currently engaged gear to another), Opening the clutch during the releasing process, causes some complexities on the setting process, which will be discussed in the following.

As shown in Figure 7.9, what is different compare to the introduced steps, in two previous refinements, is that if the clutch has been opened during the releasing process, there will be no need to synchronize the speed of the engine and the gearbox, in order to set the requested gear.

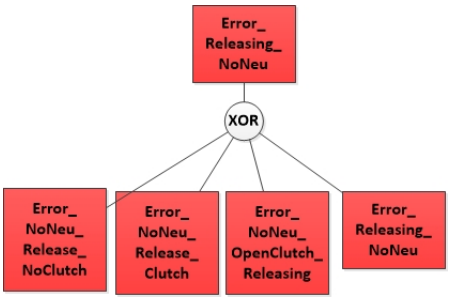
Based on the control flow refinement, the timing properties have been refined according to the patterns, explained in Sections 4.3.1 (refining deadline to sub-deadlines) and 4.3.2 (refining expiry to expiry then deadline). The concrete timing properties, replacing the abstract timing properties of the *NoNeu* scenario, are as follows:



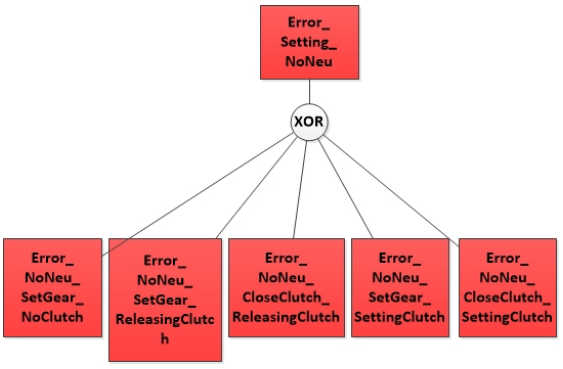
(a) Adding the required steps in order to release the current gear.



(b) Adding the required steps in order to set the requested gear.



(c) The refinement diagram of the unrecoverable errors, which may occur during the releasing process.



(d) The refinement diagram of the unrecoverable errors, which may occur during the setting process.

Figure 7.9: Adding required steps to release the currently engaged gear, and set the request gear.

Releasing Process:

$$Deadline(RequestNoNeu,\ NoNeu\_ZeroTorque \vee NoNeu\_OpenClutch\_Releasing \vee Error\_NoNeu\_OpenClutch\_Releasing, ZeroOpen\_DL)\,, \tag{7.13a}$$

$$Expiry(RequestNoNeu,\ NoNeu\_ZeroSpeed,\ Zero\_EX)\,, \tag{7.13b}$$

$$Deadline(NoNeu\_ZeroTorque,\ NoNeu\_Release\_NoClutch \vee Error\_NoNeu\_Release\_NoClutch,\ Release\_DL)\,, \tag{7.13c}$$

$$Deadline(NoNeu\_OpenClutch\_Releasing,\ NoNeu\_Release\_Clutch \vee Error\_NoNeu\_Release\_Clutch, Release\_DL)\,, \tag{7.13d}$$

Where

$$ZeroOpen\_DL \ + \ Release\_DL \ \leq R\_NN \tag{7.13e}$$

$$Zero\_EX \ + \ Release\_DL \ \leq Release\_EX \tag{7.13f}$$

$$Zero\_EX \ \leq ZeroOpen\_DL\,. \tag{7.13g}$$

Setting Process:

$$\begin{aligned} &Deadline(NoNeu\_Release\_NoClutch,\ NoNeu\_SyncSpeed \\ &\ \vee\ NoNeu\_OpenClutch\_Setting \\ &\ \vee\ Error\_NoNeu\_OpenClutch\_Setting, SyncOpen\_DL)\,, \end{aligned} \tag{7.14a}$$

$$Expiry(NoNeu\_Release\_NoClutch,\ NoNeu\_SyncSpeed,\ Sync\_EX)\,, \tag{7.14b}$$

$$\begin{aligned} &Deadline(NoNeu\_SyncSpeed,\ NoNeu\_SetGear\_NoClutch \\ &\ \vee\ Error\_NoNeu\_SetGear\_NoClutch, SetGear\_DL)\,, \end{aligned} \tag{7.14c}$$

$$\begin{aligned} &Deadline(NoNeu\_OpenClutch\_Setting,\ NoNeu\_SetGear\_SettingClutch \\ &\vee\ Error\_NoNeu\_SetGear\_SettingClutch, SetGear\_DL)\,, \end{aligned} \tag{7.14d}$$

$$\begin{aligned} &Deadline(NoNeu\_SetGear\_SettingClutch,\ NoNeu\_CloseClutch\_Setting \\ &\vee\ Error\_NoNeu\_CloseClutch\_Setting, CloseClutch\_DL),, \end{aligned} \tag{7.14e}$$

$$\begin{aligned} &Deadline(NoNeu\_Release\_Clutch, NoNeu\_SetGear\_ReleasingClutch \\ &\ \vee\ Error\_NoNeu\_SetGear\_ReleasingClutch, SetGear\_DL),, \end{aligned} \tag{7.14f}$$

$$\begin{aligned} &Deadline(NoNeu\_SetGear\_ReleasingClutch,\ NoNeu\_CloseClutch\_Releasing \\ &\vee\ Error\_NoNeu\_CloseClutch\_Releasing, CloseClutch\_DL),, \end{aligned} \tag{7.14g}$$

Where

$$SyncOpen\_DL \ + \ SetGear\_DL \ + \ CloseClutch\_DL \ \leq S\_NN\,, \tag{7.14h}$$

$$SetGear\_DL \ + \ CloseClutch\_DL \ \leq S\_NN\_RC\,, \tag{7.14i}$$

$$Sync\_EX \leq SyncOpen\_DL \ . \tag{7.14j}$$

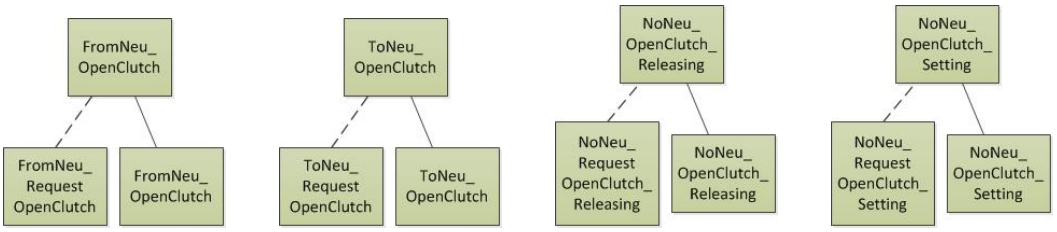
As shown above the abstract expiry on releasing process without using the clutch, has been refined by expiry (7.13b) and deadline (7.13c), based on the approach explained in Section 4.3.2. On the other hand the abstract deadlines, on the releasing and the setting processes, have been refined by concrete sequential sub-deadlines. Since the overall effect of the concrete timing properties, satisfies the abstract timing properties, the refinement is consistent.

Similar to the previous refinements, axiom 7.14j is not necessary for the refinement consistency, and it is a trivial property of the expiry property.

7.2.9 The Ninth Level of Abstraction

In this level, the existing non-deterministic choices, between opening the clutch or gaining the zero torque difference, during the releasing process, and opening the clutch or synchronizing the speed of the engine and the gearbox, during the setting process, are refined to be deterministic.

By strengthening the timing properties of the system, as explained in Section 6.4, events' occurrences can be ordered based on their occurrence times. For example in the case of gaining the zero-torque difference or opening the clutch, by having an expiry on gaining the zero-torque difference, and a delay on starting the clutch opening process, if the delay forces the opening process to start, when gaining the zero-torque difference is expiries, the non-deterministic choice between these two events, will be replaced by a deterministic choice based on time. First the zero-torque process will have the chance of accomplishment, if it has not accomplished before its expiry, it will be disabled, and the clutch opening process will be activated.



(a) Opening the clutch for the *FromNeu* scenario

(b) Opening the clutch for the *ToNeu* scenario

(c) Opening the clutch for the *NoNeu* scenario, during the releasing process

(d) Opening the clutch for the *NoNeu* scenario, during the setting process

Figure 7.10: Refining the open clutch process.

As shown in Figure 7.10, before each event, representing the accomplishment of the clutch-opening process, a new event is added, which represents the beginning of the process. Based on the concrete timing properties, if the clutch-opening process has been initiated, the alternative response (zero-torque/sync-speed) has already been expired.

Based on the system specification, during the releasing process, the controller first tries to gain the zero torque difference, and after some duration, if it has not been successful, it will give up and ask the clutch to be opened. Same scenario has been applied to the speed synchronization, before setting the request gear. The delays and expiries, added in this refinement, to enforce this behaviour, are as follows:

FromNeu Scenario:

$$Expiry(RequestFromNeu,\ FromNeu\_SyncSpeed,\ Sync\_EX)\,, \tag{7.15a}$$

$$Delay(RequestFromNeu,\ FromNeu\_RequestOpenClutch,\ OpenClutch\_Sync\_DE)\,, \tag{7.15b}$$

Where

$$Sync\_EX\ <\ OpenClutch\_Sync\_DE\ , \tag{7.15c}$$

$$OpenClutch\_Sync\_DE\ \leq\ SyncOpen\_DL\,. \tag{7.15d}$$

ToNeu Scenario:

$$Expiry(RequestToNeu,\ ToNeu\_ZeroTorque,\ Zero\_EX)\,, \tag{7.16a}$$

$$Delay(RequestToNeu,\ ToNeu\_RequestOpenClutch,\ OpenClutch\_Zero\_DE)\,, \tag{7.16b}$$

Where

$$Zero\_EX\ <\ OpenClutch\_Zero\_DE\ , \tag{7.16c}$$

$$OpenClutch\_Zero\_DE\ \leq\ ZeroOpen\_DL\,. \tag{7.16d}$$

Releasing of NoNeu Scenario:

$$Expiry(RequestNoNeu,\ NoNeu\_ZeroTorque,\ Zero\_EX)\,, \tag{7.17a}$$

$$Delay(RequestNoNeu,\ NoNeu\_RequestOpenClutch\_Releasing,\ OpenClutch\_Zero\_DE)\,, \tag{7.17b}$$

Where

$$Zero\_EX\ <\ OpenClutch\_Zero\_DE\ , \tag{7.17c}$$

$$OpenClutch\_Zero\_DE\ \leq\ ZeroOpen\_DL\,. \tag{7.17d}$$

Setting of NoNeu Scenario:

$$Expiry(NoNeu\_Release\_NoClutch,\ NoNeu\_SyncSpeed,\ Sync\_EX)\,, \tag{7.18a}$$

$$Delay(NoNeu\_Release\_NoClutch,\ NoNeu\_RequestOpenClutch\_Setting,\ OpenClutch\_Sync\_DE)\,, \tag{7.18b}$$

Where

$$Sync\_EX\ <\ OpenClutch\_Sync\_DE\ , \tag{7.18c}$$

$$OpenClutch\_Sync\_DE\ \leq\ SyncOpen\_DL\,. \tag{7.18d}$$

As shown above, for each of the events representing the accomplishment of the zero torque difference between the engine and the gearbox, there is an expiry (7.16a, 7.17a), and the corresponding clutch-opening's trigger event, has been constrained by a delay (7.16b, 7.17b), where the duration of the expiry is less than the delay (7.16c, 7.17c). It is the same case for all the events, representing the synchronization of the engine and the gearbox speeds, and their corresponding clutch-opening's trigger events.

Axioms 7.17d and 7.18d, are not necessary to prove the consistency of the refinement. They have been added, to preserve the enableness of time-progression. As explained in Section 6.3.2, if a response event has a delay longer than its deadline, it will never be enabled during the deadline period. As a result, if it is the only response event, by the end of the deadline, the time-progressing event will be disabled for ever.

Besides, the deadlines on the open clutch events have to be broken to two sequential sub-deadlines, based on the refinement pattern, explained in Section 4.3.1. The concrete deadlines are as follows:

FromNeu Scenario:

$$Deadline(RequestFromNeu,\ FromNeu\_SyncSpeed \vee FromNeu\_RequestOpenClutch,\ Sync\_DL)\,, \tag{7.19a}$$

$$Deadline(FromNeu\_RequestOpenClutch,\ FromNeu\_OpenClutch \vee Error\_FromNeu\_OpenClutch,\ OpenClutch\_DL)\,. \tag{7.19b}$$

ToNeu Scenario:

$$Deadline(RequestToNeu,\ ToNeu\_ZeroTorque \vee ToNeu\_RequestOpenClutch,\ Zero\_DL)\,, \tag{7.20a}$$

$$Deadline(ToNeu\_RequestOpenClutch,\ ToNeu\_OpenClutch \vee Error\_ToNeu\_OpenClutch,\ OpenClutch\_DL)\,. \tag{7.20b}$$

Releasing of NoNeu Scenario:

$$Deadline(RequestNoNeu,\ NoNeu\_ZeroTorque \vee NoNeu\_RequestOpenClutch\_Releasing,\ Zero\_DL)\,, \tag{7.21a}$$

$$Deadline(NoNeu\_RequestOpenClutch\_Releasing,\ Error\_NoNeu\_OpenClutch\_Releasing \vee NoNeu\_OpenClutch\_Releasing,\ OpenClutch\_DL)\,. \tag{7.21b}$$

Setting of NoNeu Scenario:

$$Deadline(NoNeu\_Release\_NoClutch,\ NoNeu\_SyncSpeed \vee NoNeu\_RequestOpenClutch\_Setting,\ Sync\_DL)\,, \tag{7.22a}$$

$$Deadline(NoNeu\_RequestOpenClutch\_Setting,\ NoNeu\_OpenClutch\_Setting \vee Error\_NoNeu\_OpenClutch\_Setting,\ OpenClutch\_DL)\,, \tag{7.22b}$$

Where

$$Zero\_DL + OpenClutch\_DL \leq ZeroOpen\_DL\,, \tag{7.22c}$$

$$OpenClutch\_Zero\_DE \leq Zero\_DL\ , \tag{7.22d}$$

$$Sync\_DL + OpenClutch\_DL \leq SyncOpen\_DL\,, \tag{7.22e}$$

$$OpenClutch\_Sync\_DE \leq Sync\_DL\ . \tag{7.22f}$$

Based on the sequential concrete sub-deadlines, during the releasing process, within *Zero_DL* time-units of the request, either the zero torque difference has been gained, or the clutch has been asked to be opened. Besides, within *OpenClutch_DL* time-units of asking the clutch to be opened, it is either opened, or it has gone to an unrecoverable error state. Based on axiom (7.22c) the timing refinement is consistent. Same thing holds for concrete sequential sub-deadlines of setting process.

Axioms such as (7.22d) and (7.22f) are important for the enableness of the response events and the *Tick_Tock* event, as explained in Chapter 6.

7.2.10 The Tenth Level of Abstraction

This refinement is the last horizontal refinement. In this level, the events of the engine, the clutch, and the gearbox are introduced, and the maximum acceptable latency of their communications with the controller, are added to the model. As it has been discussed in Section 6.4, since some new behaviours have been introduced in this refinement, their timing properties, will be added to the existing abstract timing properties of the model.

In Figure 7.11, event *Engine_Request_ZeroTorqure* represents the start of gaining the zero-torque difference process in the engine, and event *Engine_ZeroTorqure* models the accomplishment of zero-torque difference by the engine. In a similar pattern the processes in the gearbox and the clutch have been modelled by request (start of the process) and accomplishment events.

As a result, for example all the open clutch accomplishment events of the controller will be guarded by the occurrence of the event which represents the accomplishment of

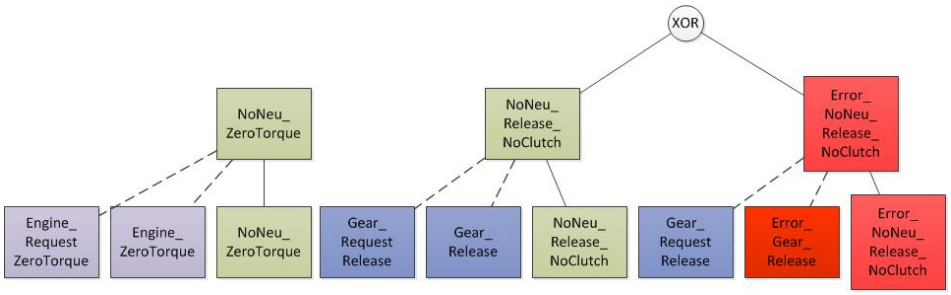


Figure 7.11: Partial refinement diagram of the tenth refinement.

the opening process in the clutch, instead of the occurrence of the previous step in the controller. Besides, each event which represents the accomplishment of a required step in the controller, triggers the next process in the corresponding component, except the accomplishment of the last step.

As shown in Figure 7.11, for a sequence of steps in the *NoNeu* scenario, *NoNeu_ZeroTorque* only happens if the *Engine_ZeroTorque* has already happened in the engine. Besides the *Gear_RequestRelease* event which represent the beginning of releasing the currently engaged gear in the gearbox, can only happen if the previous step has been accomplished in the controller (*NoNeu_ZeroTorque*).

Also, if an unrecoverable error happens in the gearbox, an unrecoverable error event will occur in the controller, too. The relation of the error events, has been provided by the timing properties. For example, since the deadline of the gearbox is less than the controller (axiom (7.29c)), after requesting neutral gear, if the controller does not hear anything from the gearbox, by the end of its deadline, it can assume that an unrecoverable error has happened there, and it will go to an error state, too. This is true for all the other steps too. The concrete timing properties of the gearbox, the clutch, the engine, and the communication channel are as follows:

Engine-Controller Communication Deadlines:

$$Deadline(RequestFromNeu, Engine\_Request\_SyncSpeed, Channel\_DL)\,, \tag{7.23a}$$

$$Deadline(NoNeu\_Release\_NoClutch,\ Engine\_Request\_SyncSpeed, Channel\_DL)\,, \tag{7.23b}$$

$$Deadline(RequestToNeu, Engine\_Request\_ZeroTorque, Channel\_DL)\,, \tag{7.23c}$$

$$Deadline(RequestNoNeu, Engine\_Request\_ZeroTorque, Channel\_DL)\,, \tag{7.23d}$$

$$Deadline(Engine\_ZeroTorque, ToNeu\_ZeroTorque \vee NoNeu\_ZeroTorque, Channel\_DL)\,, \tag{7.23e}$$

$$Deadline(Engine\_SyncSpeed, FromNeu\_SyncSpeed \vee NoNeu\_SyncSpeed, Channel\_DL)\,. \tag{7.23f}$$

Clutch-Controller Communication Deadlines:

$$Deadline(FromNeu\_RequestOpenClutch, Clutch\_Request\_Open,\ Channel\_DL)\,, \tag{7.24a}$$

$$Deadline(ToNeu\_RequestOpenClutch, Clutch\_Request\_Open,\ Channel\_DL)\,, \tag{7.24b}$$

$$Deadline(NoNeu\_RequestOpenClutch\_Releasing,\ Clutch\_Request\_Open,\ Channel\_DL)\,, \tag{7.24c}$$

$$Deadline(NoNeu\_RequestOpenClutch\_Setting,\ Clutch\_Request\_Open,\ Channel\_DL)\,, \tag{7.24d}$$

$$Deadline(FromNeu\_SetGear\_Clutch, Clutch\_Request\_Close,\ Channel\_DL)\,, \tag{7.24e}$$

$$Deadline(NoNeu\_SetGear\_ReleasingClutch, Clutch\_Request\_Close,\ Channel\_DL)\,, \tag{7.24f}$$

$$Deadline(NoNeu\_SetGear\_SettingClutch, Clutch\_Request\_Close,\ Channel\_DL)\,, \tag{7.24g}$$

$$Deadline(ToNeu\_Release\_Clutch, Clutch\_Request\_Close,\ Channel\_DL)\,, \tag{7.24h}$$

$$Deadline(Clutch\_Open, FromNeu\_OpenClutch \vee ToNeu\_OpenClutch \vee NoNeu\_OpenClutch\_Releasing \vee NoNeu\_OpenClutch\_Setting,\ Channel\_DL)\,, \tag{7.24i}$$

$$Deadline(Clutch\_Close, FromNeu\_CloseClutch \vee ToNeu\_CloseClutch \vee NoNeu\_CloseClutch\_Releasing \vee NoNeu\_CloseClutch\_Setting,\ Channel\_DL)\,. \tag{7.24j}$$

Gear-Controller Communication Deadlines:

$$Deadline(ToNeu\_ZeroTorque, Gear\_Request\_Release, Channel\_DL)\,, \tag{7.25a}$$

$$Deadline(ToNeu\_OpenClutch, Gear\_Request\_Release, Channel\_DL)\,, \tag{7.25b}$$

$$Deadline(NoNeu\_ZeroTorque, Gear\_Request\_Release, Channel\_DL)\,, \tag{7.25c}$$

$$Deadline(NoNeu\_OpenClutch\_Releasing, Gear\_Request\_Release,\ \ Channel\_DL)\,, \tag{7.25d}$$

$$Deadline(NoNeu\_SyncSpeed, Gear\_Request\_Set, Channel\_DL)\,, \tag{7.25e}$$

$$Deadline(NoNeu\_OpenClutch\_Setting, Gear\_Request\_Set,\ \ Channel\_DL)\,, \tag{7.25f}$$

$$Deadline(NoNeu\_Release\_Clutch, Gear\_Request\_Set, Channel\_DL)\,, \tag{7.25g}$$

$$Deadline(FromNeu\_SyncSpeed, Gear\_Request\_Set, Channel\_DL)\,, \tag{7.25h}$$

$$Deadline(FromNeu\_OpenClutch, Gear\_Request\_Set, Channel\_DL)\,, \tag{7.25i}$$

$$\begin{aligned} &Deadline(Gear\_Release, ToNeu\_Release\_Clutch \\ &\quad \vee\ ToNeu\_Release\_NoClutch \vee\ NoNeu\_Release\_NoClutch \\ &\quad \vee\ NoNeu\_Release\_Clutch, Channel\_DL)\,, \end{aligned} \tag{7.25j}$$

$$\begin{aligned} &Deadline(Gear\_Set, FromNeu\_SetGear\_Clutch \\ &\quad \vee\ FromNeu\_SetGear\_NoClutch\ \vee\ NoNeu\_SetGear\_NoClutch \\ &\quad \vee\ NoNeu\_SetGear\_ReleasingClutch \\ &\quad \vee\ NoNeu\_SetGear\_SettingClutch,\ \ Channel\_DL)\,. \end{aligned} \tag{7.25k}$$

Engine's Deadlines:

$$\begin{aligned} &Deadline(Engine\_Request\_SyncSpeed, Engine\_SyncSpeed \\ &\quad \vee\ \ Engine\_WaitForSyncClutch,\ \ Engine\_Sync\_DL)\,, \end{aligned} \tag{7.26a}$$

$$\begin{aligned} &Deadline(Engine\_Request\_ZeroTorque, Engine\_ZeroTorque \\ &\quad \vee\ \ Engine\_WaitForZeroClutch,\ \ Engine\_Zero\_DL)\,. \end{aligned} \tag{7.26b}$$

Clutch's Deadlines:

$$\begin{aligned} &Deadline(Clutch\_Request\_Open, Clutch\_Open \\ &\quad \vee\ \ Error\_Clutch\_Open, Clutch\_Open\_DL)\,, \end{aligned} \tag{7.27a}$$

$$\begin{aligned} &Deadline(Clutch\_Request\_Close, Clutch\_Close \\ &\quad \vee\ \ Error\_Clutch\_Close, Clutch\_Close\_DL)\,. \end{aligned} \tag{7.27b}$$

Gearbox's Deadlines:

$$Deadline(Gear\_Request\_Release, Gear\_Release \lor Error\_Gear\_Release, Gear\_Release\_DL)\,, \tag{7.28a}$$

$$Deadline(Gear\_Request\_Set, Gear\_Set \lor Error\_Gear\_Set, Gear\_Set\_DL)\,. \tag{7.28b}$$

Where

$$2 * Channel\_DL + Engine\_Zero\_DL \leq Zero\_EX\,, \tag{7.29a}$$

$$2 * Channel\_DL + Engine\_Sync\_DL \leq Sync\_EX\,, \tag{7.29b}$$

$$2 * Channel\_DL + Gear\_Release\_DL < Release\_DL\,, \tag{7.29c}$$

$$2 * Channel\_DL + Gear\_Set\_DL < SetGear\_DL\,, \tag{7.29d}$$

$$2 * Channel\_DL + Clutch\_Open\_DL < OpenClutch\_DL\,, \tag{7.29e}$$

$$2 * Channel\_DL + Clutch\_Close\_DL < CloseClutch\_DL\,. \tag{7.29f}$$

By adding the details of how long it takes that a controller's request gets to the corresponding component (*Channel_DL*), how long it takes for that component to process it (deadlines (7.26a) to (7.28b)), and how long it takes that the result gets to the controller (*Channel_DL*), and the relation of these duration with controller deadlines (axioms (7.29a) to (7.29f), the most concrete model of the gear controller system, has been provided in this level of abstraction.

7.2.11 The last Refinement and Decomposition Process

As explained in Chapter 5, in order to decompose a timed Event-B model, the variable representing the current time, plus the occurrence times, and the occurrence flags of the shared events, have to be replicated. This refinement has been dedicated to achieve this.

The model is decomposed to five components, controller, communication channel, engine, clutch, and gearbox. The interface has been considered as a part of the controller. Engine, clutch, and gearbox does not communicate with each other, and it is the controller responsibility to synchronize them. The engine, the clutch, and the gearbox, communicate with the controller through the channel. As a result, the share events between these components are as follows:

- Channel-Engine:

Engine_SyncSpeed	Engine_ZeroTorque
Engine_Request_SyncSpeed	Engine_Request_ZeroTorque

Table 7.2: This table shows the share events between the channel and the engine.

- Channel-Clutch:

Clutch_Request_Open	Clutch_Request_Close
Clutch_Open	Clutch_Close

Table 7.3: This table shows the share events between the channel and the clutch.

- Channel-Gearbox:

Gear_Request_Release	Gear_Request_Set
Gear_Release	Gear_Set
FINAL	Tick_Tock

Table 7.4: This table shows the share events between the channel and the gearbox.

- Controller-Channel:

FromNeu_SyncSpeed	FromNeu_RequestOpenClutch
FromNeu_OpenClutch	FromNeu_SetGear_NoClutch
FromNeu_SetGear_Clutch	FromNeu_CloseClutch
RequestToNeu	ToNeu_ZeroTorque
ToNeu_RequestOpenClutch	ToNeu_OpenClutch
ToNeu_Release_NoClutch	ToNeu_Release_Clutch
ToNeu_CloseClutch	RequestNoNeu
NoNeu_ZeroTorque	NoNeu_Release_NoClutch
NoNeu_RequestOpenClutch_Releasing	NoNeu_OpenClutch_Releasing
NoNeu_Release_Clutch	NoNeu_SyncSpeed
NoNeu_RequestOpenClutch_Setting	NoNeu_OpenClutch_Setting
NoNeu_SetGear_NoClutch	NoNeu_SetGear_ReleasingClutch
NoNeu_SetGear_SettingClutch	NoNeu_CloseClutch_Releasing
NoNeu_CloseClutch_Setting	

Table 7.5: This table shows the share events between the controller and the channel.

- All the components:

FINAL	Tick_Tock

Table 7.6: This table shows the share events between all the components.

As shown above, none of the error events are shared, and the *Tick_Tock* and the *FINAL* events are shared between all the components. Decomposing the *FINAL* event, the corresponding issues, and possible soloutions have been discussed in detailes, in Section 5.2.

After the decomposition, to present the possibility of refining the timing properties of each component independently, the clutch component has been refined, by adding two expiries and two delays.

7.3 Proof Statistics

Many proof obligations have been generated for the Event-B model of this case study (Table 7.7), but the important achievement is that more than 95% of them have been proved automatically. The interactively proved ones, are mostly refinement consistency POs, when an abstract timing property (mostly deadline) has been replaced by a sequence of concrete timing properties. We have elicited a systematic approach, to perform these interactive proofs.

Machine	Number of Generate PO	Automatically Proved	Automatically Proved %
m0	4	4	100
m1	156	155	99
m2	44	44	100
m3	103	103	100
m4	21	20	95
m5	130	129	99
m6	131	124	94
m7	324	291	89
m8	123	118	95
m9	64	64	100
m10	302	286	94
Total	1404	1339	95

Table 7.7: Number of generated proof obligations for each machine and how they have been proved

When a deadline has been broken to several sequential sub-deadlines, we need to show that after each step, the abstract deadline holds. We start from the first step, which can be proved automatically, then as explained in Section 4.3.1, based on the invariants which connect the occurrence of the concrete trigger events to the abstract one, all the other steps can be proved, too.

In order to explain the process in more details, we will go through the proving process of the seventh refinement (Section 7.2.8), where, releasing with or without opening the clutch is refined by its sub-steps. The concrete sequential sub-steps after receiving the request are, gaining the zero torque difference between the gearbox and the engine, or opening the clutch, and then releasing the currently engaged gear. The abstract deadline is as follow:

$$\begin{aligned} &Deadline(RequestNoNeu,\ NoNeu\_Releasing\_NoClutch \\ &\quad \vee\ NoNeu\_Releasing\_Clutch \vee Error\_Releasing\_NoNeu,\ R\_NN)\,. \end{aligned} \tag{7.30}$$

The sequential sub-deadlines are as follow:

$$\begin{aligned} &Deadline(RequestNoNeu,\ NoNeu\_ZeroTorque \\ &\quad \vee\ NoNeu\_OpenClutch\_Releasing \\ &\quad \vee\ Error\_NoNeu\_OpenClutch\_Releasing,\ ZeroOpen\_DL)\,, \end{aligned} \tag{7.31a}$$

$$\begin{aligned} &Deadline(NoNeu\_ZeroTorque,\ NoNeu\_Release\_NoClutch \\ &\quad \vee\ Error\_NoNeu\_Release\_NoClutch,\ Release\_DL)\,, \end{aligned} \tag{7.31b}$$

$$\begin{aligned} &Deadline(NoNeu\_OpenClutch\_Releasing,\ NoNeu\_Release\_Clutch \\ &\quad \vee\ Error\_NoNeu\_Release\_Clutch, Release\_DL)\,. \end{aligned} \tag{7.31c}$$

In order to prove the satisfaction of abstract deadline (7.30), by its refining deadlines, modeller has to go through the following steps:

1. Case distinction of events *NoNeu_ZeroTorque* and *NoNeu_OpenClutch_Releasing* occurrences (two cases each, occurred or not),
2. In the case where none of them have happened, based on concrete deadline (7.31a) abstract deadline is satisfied,
3. In the case where event *NoNeu_ZeroTorque* has happened:
   (a) Case distinction of event *NoNeu_Release_NoClutch* occurrence,
   (b) In the case where event *NoNeu_Release_NoClutch* has not happened, based on the invariant, which specifies the relation of event *NoNeu_ZeroTorque* occurrence time, and the request event's occurrence time, plus the concrete deadline (7.31b), the satisfaction of the abstract deadline guard can be proved,
4. In the case where event *NoNeu_OpenClutch_Releasing* has happened:
   (a) Case distinction of event *NoNeu_Release_Clutch* occurrence,

(b) In the case where event *NoNeu_Release_Clutch* has not happened, based on the invariant which specifies the relation of event *NoNeu_OpenClutch_Releasing* occurrence time, and the request event occurrence time, plus the concrete deadline (7.31c), the satisfaction of the abstract deadline guard can be proved.

In the above steps, occurrences of the error events, and the concrete releasing events have not be considered, because their occurrence will satisfy the deadline, and those cases will be discharged by the automatic provers.

7.4 UPPAAL Model of Gear Controller

So far, the specification of the gear controller case-study, and how it has been modelled in Event-B, have been explained. In this section, in order to have a better understand of the UPPAAL model of this case study, how the clutch has been modelled in [87], will be presented and briefly explained.

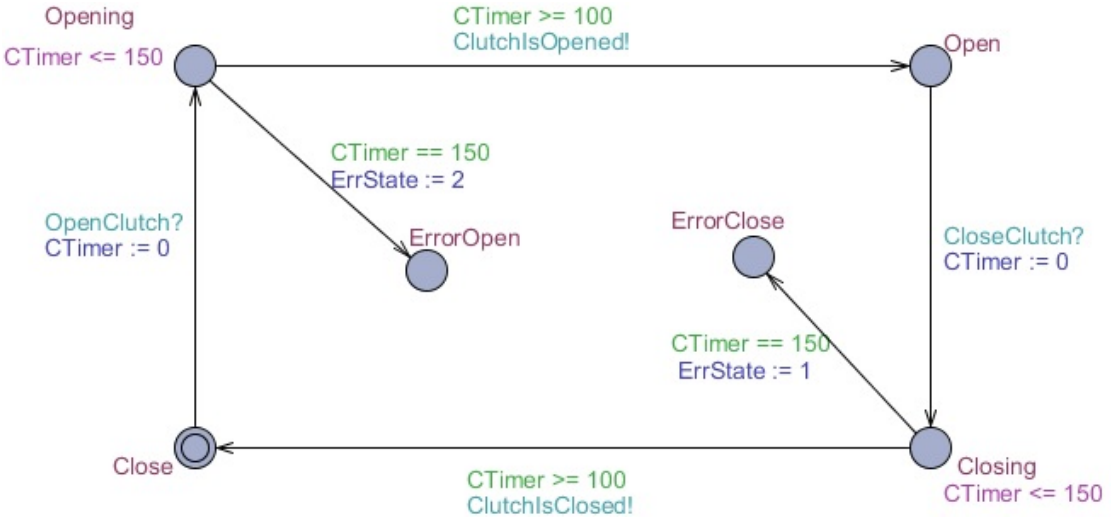


Figure 7.12: UPPAAL Model of Clutch

As shown in Figure 7.12, there are 6 operating states in the clutch. *Close* is the initial state, in which the clutch is closed and no request has been received. *Opening* is the next possible state, where an open clutch request has been received through the *OpenClutch* channel, and the clutch has started the opening process. As soon as the request is received the dedicated clock will be reset, to measure the duration of the opening process. If the clutch cannot accomplish the open state in less than 150 time-units (millisecond in here), the state's invariant will non-deterministically force one of the enabled transitions to happen. If the opening process has been accomplished before the deadline, other components will be informed through the *ClutchIsOpened* channel, and the clutch will

go to the *Open* state. A similar process will happen for the closing process, if a closing request is received in the *Open* state through *CloseClutch* channel.

As explained, the message passing, and synchronization happen through channels in an UPPAAL model, and timing properties can be specified either on states (as invariants), or on transitions (as guards).

In the model of the clutch, it is possible to check its timing properties. For example, it is possible to verify that if *CTimer* is greater than 150, the model is not in the *Opening* or the *Closing* states. This property can be verified by the following query:

$$A[]\ CTimer > 150\ imply\ not\ (Clutch.Closing\ and\ Clutch.Opening) \tag{7.32}$$

In query (7.32), *A[]* means that, the following predicate is true in all reachable states.

As shown in Figure 7.12, to model the timing properties of a state, or a transition, the timer needs to be reset. Based on the control flow of a system, and the arrangement of its timing properties, modeller has to decide how many timers are required, and where each of them has to be reset. Mostly, these decisions have to be made heuristically, since there is no specific pattern for them. On the other hand, in our approach, there is a mechanized process to encode the timing properties of a model.

This is not the only difference between these two approaches. In a timed Event-B model, durations have symbolic values, which provide a more generic model, in compare to using specific numbers (the UPPAAL's approach).

Besides, based on the refinement feature of Event-B, the consistency of the timing properties, in different levels of abstraction, can be verified in a timed Event-B model.

This example, aimed to give an idea of how the explained requirements in this chapter, can be modelled by using UPPAAL features. Besides, some of the differences between a timed Event-B model, and an UPPAAL model have been explained.

In this chapter, how the real-time automatic gear controller case-study has been modelled based on our approaches, have been explained. It was aimed to demonstrate the practicality of the introduced timing properties, and their refinement. In the next chapter, how the same principles explained in Chapter 4, can be used to model and refine the timing properties of parametrized events, will be explained.

Chapter 8

Modelling Parametrized Timing Properties In Event-B

Events may have parameters in Event-B. If either of the trigger, or the response events, are parametrized, the relation between their parameters and their timing properties, has to be specified. For example in a deadline, the sub-set of trigger event's parameters, which trigger the timing property, and the corresponding parameters of the response events, which satisfy the timing property, have to be specified.

So what will be explained in this chapter, is a generalization of the approach introduced in Chapter 4. In the following we will show how the same timing properties of Chapter 4 can be enforced to an occurrence of a trigger-response sequence for a specific parameter.

8.1 Parametrized Timing Properties Syntax

Before introducing the syntax of parametrized timing properties, the importance of the relation between parameters of trigger and response events in a trigger-response pattern, will be demonstrated through a simple example.

Assume a parametrized event T and its parametrized response event R, as shown in Figure 8.1. As represented in the model, for an occurrence of T for a parameter $x \in X$, there will be an occurrence of R for parameter $j_R(x)$, in response. j_R is a total injection function, which specifies the correlation between the parameters of T and R. For example, in transferring a message, packet by packet, a packet can be transferred if the previous one has been already transferred. In this case $j_R(x)$ will be as follows:

$$j_R \in X \rightarrowtail X\,, \tag{8.1a}$$

$$j_R(x) = x + 1\,. \tag{8.1b}$$

As a result, if a timing property is supposed to constrain the occurrence-times of parametrized trigger and response events, it should be based on their order of occurrences.

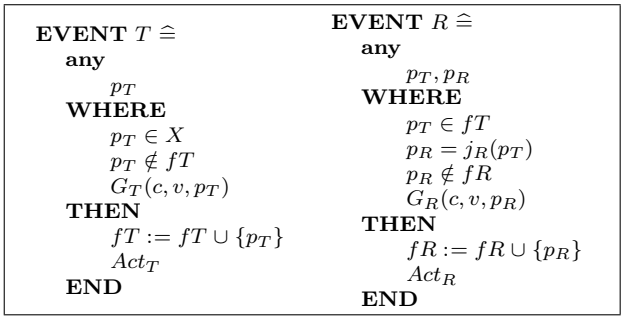


Figure 8.1: An example of a generic parametrized trigger-response pattern.

In the case of parametrized trigger-response events, in order to enforce the order between the trigger event and its possible responses, a set will be dedicated to each event, which keeps track of the parameters, the event has happened for. So when a trigger event T happens for a parameter p_T, the parameter will be added to a set T, and then set T will be used to enable the corresponding response events.

For example, a deadline of t time-units, between the occurrence of event T for a parameter $p \in X$ as the trigger, and event R for the same parameter as its response, can be specified as follows:

$$\forall p \cdot p \in X \mid Deadline(T(p), R(j_R(p)), d) \tag{8.2a}$$

$$\begin{array}{l} Where \\ \quad j_R \in X \rightarrowtail X \\ \quad \forall p \cdot p \in dom(j_R) \Rightarrow j_R(p) = p \end{array} \tag{8.2b}$$

Based on deadline (8.2a), there should not be a gap longer than d between occurrence of T and R, for a parameter p.

Based on our approach of modelling the control-flow, each parameter can only appear once, unless the occurrence set has been reset (Similar to FINAL). That is why the relation of the trigger and response parameters (j_R), has been modelled by an injection function. In the case of hierarchical iterations, such as sending several messages, part by part, the occurrence sets, should be reset by the end of each external iteration (end of a message), to provide the required initial state, for the next iteration (sending the next message).

The parametrized syntax of deadline, delay and expiry, is as follows:

$$\forall\ a\ \cdot\ P_a\ \mid\ \textbf{Deadline}\ (A(a), B_1(j_{B_1}(a)) \lor \cdots \lor B_n(j_{B_n}(a)), t)\,, \tag{8.3a}$$

$$\forall\ a\ \cdot\ P_a\ \mid \textbf{Delay}\ (A(a), B(j_B(a)), t)\,, \tag{8.3b}$$

$$\forall\ a\ \cdot\ P_a\ \mid \textbf{Expiry}\ (A(a), B(j_B(a)), t)\,, \tag{8.3c}$$

Where

$$\begin{aligned} a &\in X \\ j_B &\in X \rightarrowtail X \\ j_{B_1} &\in X \rightarrowtail X \\ &\vdots \\ j_{B_n} &\in X \rightarrowtail X \end{aligned}$$

Deadline (8.3a) means that within a duration t time-units from occurrence of event A for a parameter a, which satisfies predicate P_a, one and only one of the response events B_x ($x \in 1..n$) has to happen for a parameter $j_{B_x}(a)$. $j_{B_x}(a)$ is an injection from the parameters of the trigger event to the parameters of the response event B_x. This relation is based on the existing order of the trigger event, and its responses. Predicate P_a specifies the range of parameters, the timing property is based on.

Delay (8.3b) means that, within t time-units of the trigger event's occurrence (A), for a parameter a, which satisfies predicate P_a, response event B cannot happen for a parameter $j_B(a)$. Similar to the deadline, $j_B(a)$ is an injection from trigger event's parameters to its response's parameters based on their order.

Finally, expiry (8.3c) means that the response event, can only happen for a parameter $j_B(a)$, within t time-units of the trigger event's occurrence (A), for a parameter a, which captures predicate P_a.

In this work, there are cases of timing properties, between the occurrence of a parametrized event, for a specific parameter, and the occurrence of an unparametrized event. If the trigger event, or all of the response events, are unparametrized, the syntax will be similar to what has been presented in Section 4.2. The only difference is that the parameter value will appear in front of the corresponding event. For example a delay of t time-units, between an unparametrized trigger event A and the occurrence of a parametrized event B for a parameter b will be expressed as follows:

$$Delay(A, B(b), t)\,. \tag{8.4}$$

By using parametrized timing properties, it is possible to model timing properties on events, such as transferring a message, part by part. In the following, the semantics of parametrized timing extensions of Event-B will be discussed.

8.2 Semantics of Parametrized Timing Properties

Similar to unparametrized timing properties, based on the semantics of our parametrized timing constructs, they will be translated into Event-B variables, invariants, guards and actions.

In each case we assume there is already an order between the occurrence of the trigger event and the corresponding response events, for their parameters. The assumption is that for a response event to happen for a parameter, there is one and only one parameter in the trigger event, which has to happen before hand. As shown in Figure 8.1, event A add the current parameter to set A as one of its actions. So when a parameter x belongs to A, indicates that event A has already happened for it. Also, one of the response events' guards, checks the occurrence of trigger event A for the corresponding parameter, by evaluating its occurrence set.

It has not been assumed that the trigger and response events will occur only once for each parameter. Similar to what has been explained in Chapter 4, by having an event by the of each iteration, which resets all the corresponding occurrence sets (the $FINAL$ event), iterative behaviours can be modelled.

8.2.1 Semantics of Parametrized Delay and Expiry

To give an Event-B semantics to a delay on a parametrized trigger-response pattern, the variable which records the trigger event occurrence times, is a function from the event's parameters to their occurrence times. Besides, the trigger and the response events, have occurrence sets, which are used to enforce the order. A generic parametrized trigger-response pattern with a delay property is shown in Figure 8.2(a) and Figure 8.2(b) presents how the delay property is enforced by using standard Event-B constructs.

In Figure 8.2, X specifies the range of the parameters of A, and P_a specifies the range of its parameters, the timing property is concerned with. As shown in Figure 8.2(b), event B can only occur for a parameter $j_B(a)$, if event A has already happened for parameter a.

Similar to what we have in the unparametrized semantics, there are two invariants in the parametrized delay semantics. $inv1$ expresses the order of trigger and response events, and $inv2$ express the property that if the response has happened for a parameter

TIMING
$\forall a \cdot P_a \mid$
$Delay(A(a), B(j_B(a)), t)$
EVENT $A \widehat{=}$
any
a
WHERE
$a \in X$
$a \notin fA$
$G_A(c, v, a)$
THEN
$fA := fA \cup \{a\}$
Act_A
END
EVENT $B \widehat{=}$
any
a, b
WHERE
$a \in fA$
$b = j_B(a)$
$b \notin fB$
$G_B(c, v, b)$
THEN
$fB := fB \cup \{b\}$
Act_B
END

(a) Event A and B plus a delay

INVARIANTS
(inv_1) $\forall a \cdot P_a \wedge a \notin fA \Rightarrow$
$j_B(a) \notin fB$
(inv_2) $\forall a, b \cdot b \in fB \wedge$
$a \in j_B^{-1}(b) \wedge P_a \Rightarrow$
$tB(b) \geq tA(a) + t$
EVENT $A \widehat{=}$
any
a
WHERE
$a \in X$
$a \notin fA$
$G_A(c, v, a)$
THEN
$fA := fA \cup \{a\}$
$tA(a) := time$
Act_A
END
EVENT $B \widehat{=}$
any
a, b
WHERE
$a \in fA$
$b = j_B(a)$
$b \notin fB$
$P_a \Rightarrow time \geq tA(a) + t$
$G_B(c, v, b)$
THEN
$fB := fB \cup \{b\}$
$tB(b) := time$
Act_B
END
EVENT $Tick\_Tock \widehat{=}$
any
$tick$
WHERE
$tick > 0$
THEN
$time := time + tick$
END

(b) Encoded delay for events A and B.

Figure 8.2: Semantics of a parametrized delay property in Event-B.

included in the delay, its occurrence time for that parameter must exceed the occurrence time of trigger event for the corresponding parameter by at least t.

The consistency proof will be similar to what has been explained in Section 4.2.1. For the Rodin model of the delay semantics, presented in Figure 8.2, 20 POs were generated, from which 18 were discharged automatically and the rest were discharged interactively.

$$\forall\ a\ \cdot\ P_a\ \mid\ Expiry(A(a), B(j_B(a)), t) \tag{8.5}$$

Semantics of the expiry is similar to the delay semantics. On a same generic trigger-response example, shown in Figure 8.2(a), if we want to have expiry (8.5) instead of the delay, the timing guard in the response event, in Figure 8.2(b), will be changed to the following predicate:

$$P_a\ \Rightarrow\ time\ \leq\ tA(a)\ +\ t \tag{8.6}$$

And the invariants will be as follows:

$$\forall a\ \cdot\ P_a \wedge a \notin fA\ \ \Rightarrow\ \ j_B(a) \notin fB \tag{8.7}$$

$$\forall a, b\ \cdot\ b \in fB \wedge a \in j_B^{-1}(b) \wedge P_a\ \ \Rightarrow\ \ tB(b) \geq time + tA(a) \tag{8.8}$$

Similar to the delay semantics, invariant (8.7) expresses the order of trigger and response events, and invariant (8.8) expresses the expiry property. In the Rodin development of the expiry semantics 20 POs were generated, where 18 have been discharged automatically, and the reset have been discharged interactively.

Based on guard (8.6), if the trigger event's parameter, satisfies predicate P_a, then the response event may be enabled, if its expiry duration has not been passed.

8.2.2 Semantics of Parametrized Deadline

To explain the semantics of deadline for a parametrized trigger-response pattern, a generic trigger-response with n ($n > 0$) alternative responses, will be explained in the following.

TIMING
$\forall a \cdot P_a \mid$
$Deadline(A, B_1(j_{B_1}(a)) \vee .. \vee B_n(j_{B_n}(a)), t)$
EVENT $A \mathrel{\widehat{=}}$
any
a
WHERE
$a \in X$
$a \notin fA$
$G_A(c, v, a)$
THEN
$fA := fA \cup \{a\}$
Act_A
END
EVENT $B_x \mathrel{\widehat{=}}$
any
a, b
WHERE
$a \in fA$
$b = j_{B_x}(a)$
$j_{B_1}(a) \notin fB_1$
$\vdots$
$j_{B_n}(a) \notin fB_n$
$G_{B_x}(c, v, b)$
THEN
$fB_x := fB_x \cup \{b\}$
Act_{B_x}
END

(a) Event A and B_x ($x \in 1..n$) plus a deadline. X is the range of events A parameters.

INVARIANTS
(inv_1) $\forall a \cdot P_a \wedge a \notin fA \Rightarrow$
$j_{B_1}(a) \notin fB_1 \wedge \cdots \wedge$
$j_{B_n}(a) \notin fB_n$
(inv_2) $\forall\ a\ \cdot P_a \wedge a \in fA \wedge$
$j_{B_1}(a) \notin fB_1 \wedge .. \wedge$
$j_{B_n}(a) \notin fB_n \Rightarrow$
$time \leq tA(a) + t$
(inv_3) $\forall a, b \cdot b \in fB_1 \wedge$
$a \in j_{B_1}^{-1}(b) \wedge P_a \Rightarrow$
$tB_1(b) \leq tA(a) + t$
$\vdots$
(inv_{n+2}) $\forall a, b \cdot b \in fB_n \wedge$
$a \in j_{B_n}^{-1}(b) \wedge P_a \Rightarrow$
$tB_n(b) \leq tA(a) + t$
EVENT $A \mathrel{\widehat{=}}$
any
a
WHERE
$a \in X$
$a \notin fA$
$G_A(c, v, a)$
THEN
$fA := fA \cup \{a\}$
$tA(a) := time$
Act_A
END
EVENT $B_x \mathrel{\widehat{=}}$
any
a, b
WHERE
$a \in fA$
$b = j_{B_x}(a)$
$j_{B_1}(a) \notin fB_1$
$\vdots$
$j_{B_n}(a) \notin fB_n$
$G_{B_x}(c, v, b)$
THEN
$fB_x := fB_x \cup \{b\}$
Act_{B_x}
END
EVENT $Tick\_Tock \mathrel{\widehat{=}}$
any
$tick$
WHERE
$tick > 0$
$\forall\ a\ \cdot P_a \wedge a \in fA \wedge$
$j_{B_1}(a) \notin fB_1 \wedge .. \wedge$
$j_{B_n}(a) \notin fB_n \Rightarrow$
$time + tick \leq tA(a) + t$
THEN
$time := time + tick$
END

(b) Encoded deadline for Event A and B_x.

Figure 8.3: Semantics of parametrized deadline in Event-B.

As shown in Figure 8.3(a), a response event B_x can only happen for a parameter $j_{B_x}(a)$, if A has already happened for parameter a. The parametrized deadline of Figure 8.3(a), is based on the parametrized trigger event (A), and its possible parametrized responses ($B_1..B_n$). If event A has happened for a parameter a, which satisfies predicate P_a, then the guard on the $Tick\_Tock$ event, enforces one of the response events B_x ($x \in 1..n$) to happen, for parameter $j_{B_x}(a)$, before passing the deadline duration.

Similar to unparametrized deadline semantics explained in Section 4.2.3, inv_1 expresses the order of trigger and response events, inv_2 expresses the property of the current time value, when the trigger event has happened, but no response event has happened yet. Finally invariants $inv_3..inv_{n+2}$ express the property that, if a response event B_x happens for a parameter included in the deadline, its occurrence time will not exceed the occurrence time of A for the corresponding parameter by more than t.

For a Rodin development of a deadline as follows:

$$Deadline(A, B_1(j_{B_1}(a)) \vee B_2(j_{B_2}(a)), t) \tag{8.9}$$

38 proof obligations were generated from which 2 were discharged interactively and the rest were discharged automatically. The consistency proof is similar to the unparametrized deadline explained in Section 4.2.3.

8.3 Some Patterns to Refine Parametrized Timing Properties

All the refinement patterns, introduced in Section 4.3 for unparametrized timing properties, can be used for the parametrized one, with some changes.

In order to apply those patterns on parametrized timing properties, instead of occurrence flags, the occurrence sets should be used, and all the occurrence-time variables, will be relations from the parameters of the corresponding event to their occurrence time. In the following we go through some of the main ones, mostly used in our case-study.

Besides, in Section 8.3.2 how an unparametrized deadline can be refined by an iterative parametrized deadline will be explained. Unlike the unparametrized timing properties, it is possible to have the same event as the trigger and the response events, in a parametrized one, where the occurrence of that event for a parameter, triggers its occurrence for another parameter. Based on this feature it is possible to have iterative parametrized timing properties, on iterative parametrized events.

8.3.1 Refining a Parametrized Deadline to Sequential Parametrized Sub-Deadlines

Consider an abstract model of a system where there is a deadline between event A and event B. As shown in Figure 8.4, event B can only occur for a parameter b, if event A has already happened for it. The deadline property of this level of abstraction, is shown in Figure 8.5(a).

As shown in Figure 8.4, event B has been broken into two sequential steps, in the refinement. By breaking event B to B_1 followed by B_2, its related deadline needs to be broken too. Similar to pattern 4.3.1, the abstract event has been refined by the second step, and the first step refines *skip*.

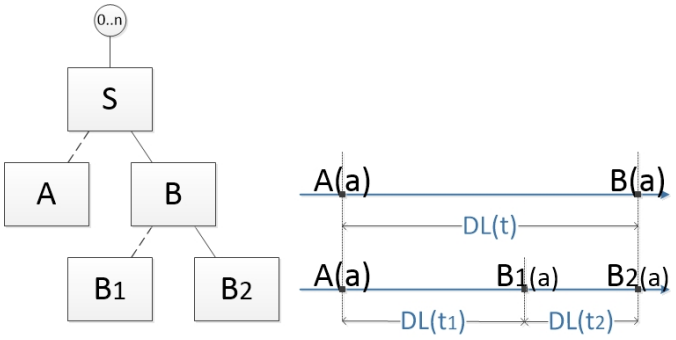


Figure 8.4: Refining an abstract paramtrized deadline to two parametrized sub-deadlines, is presented by the refinement diagram on the left. *DL(x)* presents a deadline property with a period of x in the timing diagrams

Now, in order to respond to the trigger event, two steps have to be accomplished, where each of them has its own deadline. In the concrete level, the trigger event of the deadline property for event B_1 is event A and the trigger event for the deadline of event B_2 is event B_1. Hence, the abstract deadline should be broken into two new deadlines, in a way that their combination, based on the concrete order, does not violate the abstract deadline ($t_1 + t_2 \leq t$).

As shown in Figure 8.5, in the concrete machine, the abstract deadline between event A and event B is refined by the following deadlines:

$$\forall a \cdot a \in X \mid Deadline(A(a), B_1(a), t_1)\,, \tag{8.10a}$$

$$\forall b \cdot b \in X \mid Deadline(B_1(b), B_2(b), t_2)\,. \tag{8.10b}$$

As shown in Figure 8.5(c), based on the guard of deadline (8.10b) on the $Tick\_Tock$ event, if event B_1 has happened for a parameter b, and event B_2 has not happened for it yet, then the current value of time should be less than or equal to the occurrence time of event B_1 for parameter b, plus the deadline period (t_2). By having this timing property the relation between the occurrence time of event B_2 and event B_1 has been specified.

Based on the same principle explained in refinement pattern 4.3.1, we need to specify the relation of the occurrence time of event B_2 and event A, by specifying the relation of the occurrence-time of events B_1 and A.

The gluing invariants required to prove the consistency of this refinement, are as follows:

TIMING
$\forall a \cdot a \in A \mid$
$Deadline(A(a), B(a), t)$
EVENT $A \widehat{=}$
any
a
WHERE
$a \in X$
$a \notin fA$
$G_A(c, v, a)$
THEN
$fA := fA \cup \{a\}$
Act_A
END
EVENT $B \widehat{=}$
any
a, b
WHERE
$a \in fA$
$b = a$
$b \notin fB$
$G_B(c, v, b)$
THEN
$fB := fB \cup \{b\}$
Act_B
END

(a) Events A and B

TIMING
$\forall a \cdot a \in A \mid$
$Deadline(A(a), B_1(a), t_1)$
$\forall b \cdot b \in B_1 \mid$
$Deadline(B_1(b), B_2(B), t_2)$
EVENT $A \widehat{=}$
any
a
WHERE
$a \in X$
$a \notin fA$
$G_A(c, v, a)$
THEN
$fA := fA \cup \{a\}$
Act_A
END

EVENT $B_1 \widehat{=}$
any
a, b
WHERE
$a \in fA$
$b = a$
$b \notin fB_1$
$G_{B_1}(c, v, b)$
THEN
$fB_1 := fB_1 \cup \{b\}$
Act_{B_1}
END

EVENT B_2 **refines** $B \widehat{=}$
any
b_1, b_2
WHERE
$b_1 \in fB_1$
$b_2 = b_1$
$b_2 \notin fB_2$
$G_{B_2}(c, v, b_2)$
THEN
$fB_2 := fB_2 \cup \{b_2\}$
Act_{B_2}
END

(b) Events A, B_1 and B_2

EVENT $Tick\_Tock \widehat{=}$
any
$tick$
WHERE
$tick > 0$
$\forall a \cdot a \in fA \wedge a \notin fB \Rightarrow$
$time + tick \leq tA(a) + t$
THEN
$time := time + tick$
END

EVENT $Tick\_Tock$ **refines** $Tick\_Tock \widehat{=}$
any
$tick$
WHERE
$tick > 0$
$\forall a \cdot a \in fA \wedge a \notin fB_1 \Rightarrow$
$time + tick \leq tA(a) + t_1$
$\forall b \cdot b \in fB_1 \wedge b \notin fB_2 \Rightarrow$
$time + tick \leq tB_1(b) + t_2$
THEN
$time := time + tick$
END

(c) Event $Tick\_Tock$ in abstract and concrete machines.

Figure 8.5: Events A and B plus their deadline property in the abstract Machine in 8.5(a), followed by event A , events B_1 and B_2 in the concrete machine plus their concrete timing properties in 8.5(b).

- The relation between the abstract event and its refining event (fB_2 and fB are the occurrence sets of the corresponding events):

$$fB_2 = fB\,, \tag{8.11}$$

- The order between the concrete events:

$$\forall b \cdot b \in fB_1 \Rightarrow b \in A\,, \tag{8.12}$$

- The relation between the abstract trigger event's occurrence-times, and the occurrence-times of the concrete trigger events:

$$\forall b \cdot b \in fB_1 \Rightarrow tB_1(b) \leq tA(b) + t_1\,, \tag{8.13a}$$

$$\forall b \cdot b \in fA \wedge b \notin fB_1 \Rightarrow time \leq tA(b) + t_1\,. \tag{8.13b}$$

In the above invariants, tX is a total function from the occurrence-set of event X to its occurrence-times. Invariant (8.11) specifies that the occurrence of event B_2 for a parameter, is equivalent to the occurrence of event B for that parameter.

The relation of the occurrence time of event B_1 and event A has been specified by the gluing invariant (8.13a) based on deadline (8.10a). Based on invariant (8.13a) we know that if B_1 has occurred for a parameter b, event A had happened at most t_1 times ago for b.

Invariant (8.13b) which is equivalent to the required guard on the $Tick\_Tock$ event for deadline (8.10a) provides the required information to discharge the corresponding POs of invariant (8.13a).

For the Rodin development of this refinement pattern, 23 POs were generated for the abstract machine, which all were discharged automatically, and 67 POs were generated for the concrete machine, from which only one needed interactive prove.

It should be mentioned that the abstract deadline can be broken into more than two sub-deadlines either by successive refinement steps or by refining the abstract event with more than two sub-sequential events in one refinement step. For these refinement cases, it will be possible to follow a similar approach.

8.3.2 Refining An Abstract Deadline to An Iterative Sub-Deadline

Sometimes there is an iterative event in a system, which happens for a finite number of times, and completes a task gradually. This behaviour usually has some properties, related to the overall effects of the iterations, and some properties, focused on each iteration. For example, transferring a message part by part in a system, may have some overall timing properties, and some timing properties on transferring each part.

In stepwise modelling and reasoning, one possible way of dealing with these types of behaviour, is to abstract them by two events, representing the start and the end of an iterative behaviour. As a result, the overall properties can be verified for these two events, and then by adding the iterations in a refinement, it will be possible to express and verify, the properties of each iteration, and their consistency with those abstract overall properties (similar to the loop correctness [71] in Hoare logic).

From timing point of view, we will be able to verify the consistency of each occurrences deadline, with the overall deadline of the task accomplishment In this way, the overall timings of a system is abstract, and it is refined by timing properties on parts.

Assume a case where in the abstraction, we have a trigger event A and its response event B, which has to happen within t time-units of its trigger's occurrence. Then, a new iterative event B_1, with n times iteration, will be added in a refinement. Event B_1

represents the required pre-steps of abstract event B, and event$B$ will be refined by event B_2, based on the new order (Figure 8.6).

In the refinement diagram, presented in Figure 8.6, $0..n-1$ in the circle represents the fact that event B_1 has to happen n times before event B can happen.

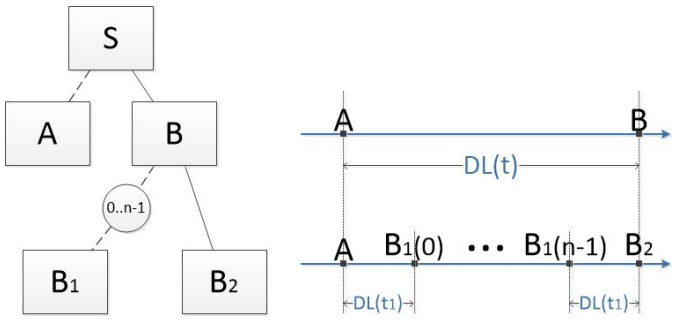


Figure 8.6: Refining a deadline to an iterative deadline.

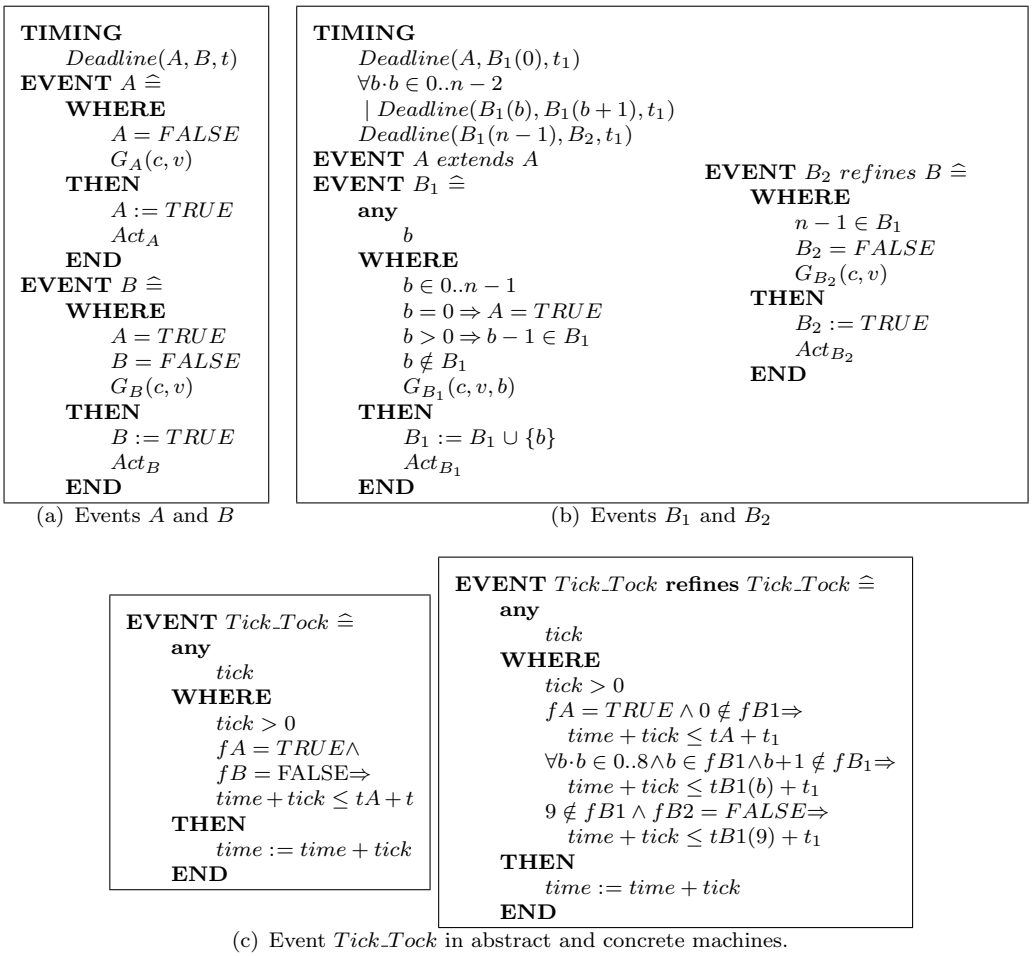
Based on the concrete order of events, the abstract deadline will be refined by a sequence of concrete deadlines, as follows (Figure 8.7):

- A deadline of t_1 time-units, between the trigger event (A) and the first occurrence of the iterative event (B_1),
- A deadline of t_1 time-units, between each occurrence of the iterative event and the next one,
- A deadline of t_1 time-units, between the last occurrence of the iterative event and occurrence of event B_2,

For the sequence of the concrete deadlines, to satisfy the abstract deadline, the sum of their durations, should not be exceeding the abstract deadline's duration ($t = (n+1)*t_1$).

As shown in Figure 8.7(b), the deadline between occurrence of A and occurrence of B_1 for parameter *0*, and the deadline between occurrence of B_1 for parameter $n-1$ and occurrence of B_2, are timing properties between an unparametrized event and a single occurrence of a parametrized one. Consequently, as explained at the beginning of this chapter, their syntax is a combination of the parametrized and the unparametrized syntaxes.

To prove the consistency of the timing properties' refinement, we need to show that, based on the iterative deadline, n times occurrence of B_1, will not take more than $(n-1)*t_1$ time-units, which combined with the deadline duration, between occurrence of A and the first occurrence of B_1, and the maximum gap between the last occurrence of B_1 and the occurrence of B_2 ($(n-1)*t1+t1+t1$), will not exceed the abstract



(a) Events A and B

(b) Events B_1 and B_2

(c) Event *Tick_Tock* in abstract and concrete machines.

Figure 8.7: Events A and B plus their deadline property in the abstract machine presented in 8.7(a), followed by events A , B_1, and B_2, in the concrete machine, accompanied by their concrete timing properties in 8.7(b).

deadlines duration. To prove this, we need to prove the following invariants for the occurrence of event B_1 for a parameter $b \in 0..n - 1$:

$$\forall\, b \cdot b \in B_1 \land b > 0 \Rightarrow b - 1 \in B_1\,, \tag{8.14a}$$

$$\forall\, b \cdot b \in B_1 \Rightarrow 0..b \subseteq B_1\,. \tag{8.14b}$$

Invariants (8.14a) and (8.14b) show the sequential order of the iterative occurrences. Based on (8.14b), the occurrence of B_1 for a parameter, applies that it has already happened for all the previous parameters.

Based on the ordering invariants and the timing properties, it is possible to express the overall effects of the concrete timing properties as follows:

$$A = TRUE \land 0 \in B_1 \Rightarrow B_1 t(0) \leq tA + t1\,, \tag{8.15a}$$

$$A = TRUE \land 0 \notin B_1 \Rightarrow time \leq tA + t1\,, \tag{8.15b}$$

$$\forall\ b \cdot b < n \ \wedge\ b \in B_1 \ \wedge\ b+1 \notin B_1 \Rightarrow\ time \leq tB_1(b) + t_1\,, \quad (8.15c)$$

$$\forall\ b \cdot b \geq 0 \ \wedge\ b \in B_1 \ \Rightarrow\ tB_1(b) \leq tB_1(0) + ((b) * t_1)\,. \quad (8.15d)$$

Invariant (8.15a) connects the occurrence time of event A (abstract trigger), to the first occurrence of the iterative event, and invariant (8.15b) is required to discharged its corresponding proof obligations. Invariant (8.15c) is the iterative deadline, and invariant (8.15d) is what can be proved based on invariants (8.15c) and (8.14b).

By having invariants (8.15d) and (8.15a), we can prove that the concrete deadlines are consistent with their abstract one. This proof needs to be done interactively, by breaking the possible cases in three groups as follows:

1. A happened but B_1 has not happened for any parameter yet,
2. B_1 happened for a parameter $x < n-1$ but has not happened for $x+1$,
3. B_1 happened for $n-1$ but B_2 has not happened yet.

Invariant (8.15a) is required for case 1. Both Invariants (8.15d) and (8.15a) are required for cases 2 and 3.

By using this approach, it is possible to model an iterative event, in two levels of abstraction, and verify the satisfaction of its overall accomplishment's deadline, based on our assumptions of each occurrence.

In the Rodin development of this refinement pattern, 14 POs were generated for the abstract machine, and all of them were discharged automatically. For the concrete machine, 69 POs were generated from which 63 were discharged automatically, and the rest were discharged interactively.

8.4 Decomposition of Parametrized Timing Properties

By applying the same approach, explained in Chapter 5, a model with parametrized timing properties can be decomposed. Similar to the unparametrized timing properties, in a parametrized model, the occurrence sets and occurrence-time relations of shared events have to be replicated.

8.5 Achievements

In this chapter how deadline, delay, and expiry, can be used to specify the timing properties of parametrized events has been explained. Besides, a new syntax has been introduced, in order to express the parametrized timing properties, and how we have encoded

them in Event-B were explained. In the end some patterns to refine the parametrized timing properties, based on some generic control flow refinement patterns, have been explained.

In the following Chapter, a message passing case-study will be explained. This case-study have been designed to evaluate the practicality of parametrized timing properties and their refinement patterns.

Chapter 9

Message Passing Case-study

This case-study is designed to investigate the practicality of the parametrized timing properties, their refinement patterns, and decomposition. The corresponding system of this case-study, consists of a sender, a receiver and a two-way channel between them. The goal is to transfer a fixed size message from sender to receiver, piece by piece.

In respect of demonstrating the practicality of our approach, this case-study has several advantages. Despite its simplicity, it covers iterative timing properties, timing properties between unparametrized and parametrized events, and parametrized timing properties.

In the following, the functionality of each component, involved in this case-study, is briefly explained.

- **Sender**
  A message is partitioned to several packets. Each packet has two parts, a unique id, and data. There is an incremental order between ids. The sender is responsible for sending the message, packet by packet, through the channel, and keeping track of the transferred packets. Besides, it will resend a packet, if it has not received the packet's acknowledgement, within a specific period of time.

- **Receiver**
  The receiver listens to the other side of the channel, to collect the arriving packets, and send back their acknowledgements to the sender. In the case of a redundant packet, just its acknowledgement will be sent.

- **Channel**
  The channel connects the sender and the receiver. At any given time, a specific amount of data can be transferred by the channel, and the received data will be removed from the channel. In order to simplify the case-study, the channel's capacity is assumed to be a packet of data. Besides, packets may be lost by the channel, and do not get to their destinations.

An overview of the system has been presented so far. In the following the requirements of this case-study will be discussed in details.

9.1 Requirements of the Message Passing Case-study

The requirements of the message passing case-study are categorized in four groups, environment assumptions, performance, functional, and error detection requirements. Since our focus is the modelling of timing properties, there will be more emphasis on the time-related requirements.

To have a more generic model, instead of real values, symbolic values have been used as the timing properties' durations. Some of the requirements, specify the existing relations, between these symbolic values.

9.1.1 Environment Assumptions

These assumptions specify the behaviour, and the properties of the channel, which is our environment in this system. These assumptions are critical for developing the right controller.

EA1. It should be possible to use the channel to transfer a packet,

EA2. Packets may be lost by the channel,

EA3. Transferring a packet should not take more than *ChannelDL* time-units, if it has not been lost.

9.1.2 Functional

The functional requirements specify the desirable behaviours of the sender and the receiver.

F1. A message sent by the sender should be received by the receiver, unless the connection is faulty,

F2. It should be possible to use the sender to send a message,

F3. It should be possible to use the receiver to receive the packets, sent by the sender,

F4. It should be possible to use the receiver to send the acknowledgement of the received packets,

F5. An acknowledgement packet has to have the same id as its corresponding received packet,

F6. Each acknowledgement packet has to be marked by *ACK*,

F7. The packets has to be sent in an incremental order, starting by the first packet,

F8. The sender should not send the next packet, if the acknowledgement of the previous one is yet to be received,

F9. It should be possible to use the sender to resend a message,

F10. A packet should not be resent more than *MaxResend* times ($MaxResend > 1$),

F11. The receiver should resend the acknowledgement, if it receives a packet which has been already received,

F12. End of a message has to be marked by a *FIN* packet,

F13. The sending process is over by receiving the acknowledgement of *FIN*,

9.1.3 Performance

The performance requirements specify the tolerable latencies of the processes of the sender and the receiver.

P1. A successful transferring process of a message, consists of *last+1* packet, should be accomplished within *DataTDL* time-units ($DataTDL \geq (last+1) * PacketTDL$),

P2. The process of transferring a packet (from the first sending attempt, to receiving the acknowledgement), should not take more than *PacketTDL* ($PacketTDL \geq (MaxResend+1) * ResendingDL$), if the packet or its acknowledgement have not been lost,

P3. A packet acknowledgement should be received within $2*ChannelDL$ time-units of its sending, unless either of the sent packet or its acknowledgement is lost,

P4. A packet must be resend within *ResendingDL* time-units ($ResendingDL > 2*ChannelDL$), if its acknowledgement has not been received, and no unrecoverable error has happened in the sender,

P5. A packet can be resend if at least $2*ChannelDL$ time-units is passed from the previous attempt,

P6. The next packet has to be sent at the same time as the previous packet acknowledgement is arrived,

P7. An acknowledgement has to be sent at the same time as its packet is arrived,

9.1.4 Error Detection

The error detection requirements specify the unrecoverable error states of the sender and the receiver.

E1. When the sender has not received the acknowledgement of the last transferring attempt of a packet, within its deadline (δ),

E2. When the receiver has not received the next packet, within $MaxResend * \delta + 2 * \alpha$ time-units of the previous one,

In Figure 9.1 the sequence of processes to accomplish, in order to transfer a packet has been presented. The transferring process of a packet starts by sending it, and finishes by receiving its acknowledgement, or the occurrence of an unrecoverable error.

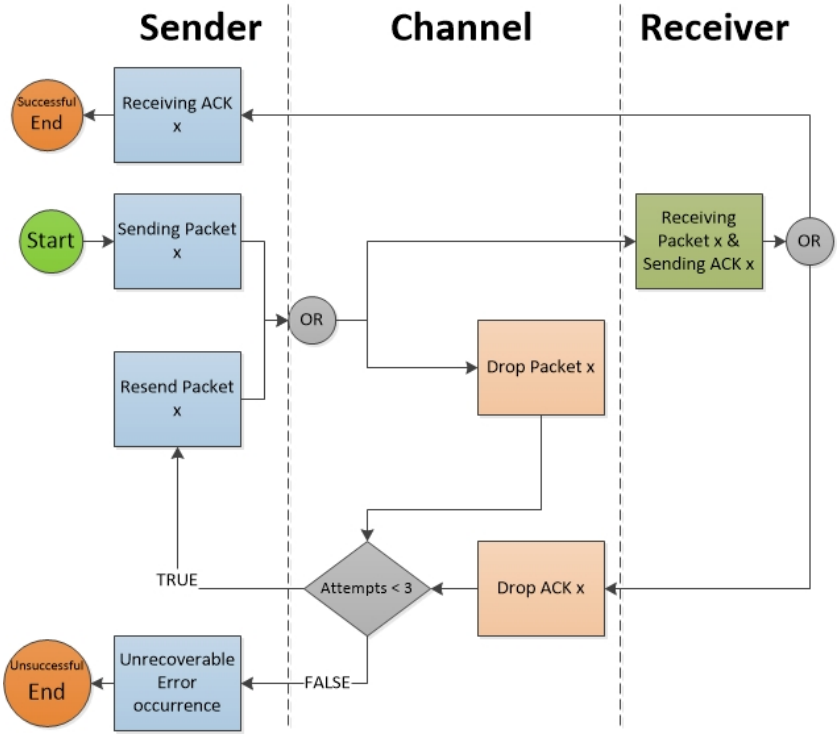


Figure 9.1: Control Flow Diagram of Sending a Packet

Since, each component has no direct awareness about the state of the others, they synchronize based on time and message passing. How the timing properties of the system (Sections 9.1.3, 9.1.4, and 9.1.1) help to synchronize these components, is explained briefly in the following:

- After sending a packet, sender wait as long as it takes for the packet to be transferred, plus the duration required for its acknowledgement to arrive ($\alpha + \alpha$). If no acknowledgement has arrived within this period, the packet will be resend,

- Same process will be repeated for each resend, except the final one,
- After the final resend, if the sender has not received the acknowledgement within $2 * \alpha$, it assumes a faulty connection and goes to an error state,
- When a packet is received by the receiver, and it is not *FIN* (the last packet), if the receiver does not receive anything, within the required duration to transfer its acknowledgement back to the sender (α), plus the duration required to receive a packet, which has been tried to be sent by the sender, for *MaxResend+1* times ($((MaxResend - 1) * \delta) + \alpha$), the receiver will assume a faulty connection, and goes to an error state.

As explained, the combination of timing properties, enforces the desirable behaviour of the system. For example, by considering how long it takes for a packet to be transferred through the channel, and how soon the receiver sends the acknowledgement after receiving a packet, the sender knows how long it has to wait before resending a packet.

9.2 Refinement Strategy

So far the requirements of the case-study have been introduced. In this section our strategy of including those requirements in the model, step by step, by each refinement will be explained.

In the most abstract level, $P1$, $F1$, $F6$, and $F12$ have been included, by having three events, representing the start of transferring process by the sender, successful transfer, and error occurrence, plus their timing property. Requirements $F6$ and $F12$ have been included in the context, since the characteristics of a message have been explained in terms of axioms. The first refinement explained in Section 9.3.2, adds $F7$, and $F13$ to the model by adding an iterative event, representing the packet by packet transferring of data. In the next refinement explained in Section 9.3.3, $P2$ has been added to the model, by adding the timing properties of the iterative behaviour. In the third refinement explained in Section 9.3.4, $F2$ has been added by introducing the iterative sending event. As it will be explained in Section 9.3.5, the forth refinement adds $F9$, $F10$, and $P4$ to the model, by introducing the resending of a lost packet to the model. In the fifth refinement, explained in Section 9.3.6, requirements $EA1$ and $EA2$ will be added, by introducing the loss of data in the channel. The sixth refinement explained in Section 9.3.6, requirements $F3$, $F4$, $F5$, $F8$, $F11$,$P3$, $P5$, $P6$, $P7$, $E1$, $E2$, and $EA3$ have been added to the model, by introducing the receiver's events and their timing properties.

The final two refinement have been dedicated to provide the required changes to decomposed a timed Event-B model as explained in Chapter 5.

In the following, based on the explained requirements and properties of the system, we will explain how the system is modelled, step by step in Event-B. The main focus will be on the process of modelling, refining and decomposing the timing properties of this case-study.

9.3 Event-B Model of the Message Passing System

For the clarity of the case-study, a fixed size message is assumed. The structure of a message is modelled in the context, and properties such as the order of the ids, are specified as axioms. The Event-B model of the case-study consists of 10 machines and 4 contexts. In the following, how the model is developed gradually, and how the abstract timing properties are replaced by the concrete ones, will be explained in details.

9.3.1 The Most Abstract Machine and Context

As mentioned, the construct of messages is specified in the context. The most abstract context is mostly dedicated to this.

An Event-B Specification of c00
Creation Date: 6Nov2012 @ 00:28:39 PM

CONTEXT c00

SETS

$NET\_MESSAGE$

CONSTANTS

$Data$

ACK

FIN

$Send\_Message$

$last$ Last sequence number of sendable message

$DataTDL$ Deadline duration to transfer Send_Message

AP_DATA Application Data

AXIOMS

axm1 : $partition(NET\_MESSAGE, Data, \{ACK\}, \{FIN\})$

axm2 : $AP\_DATA \subseteq Data$

axm3 : $Send\_Message \in \mathbb{N} \pfun NET\_MESSAGE$

axm4 : $ran(Send\_Message) = Data \cup \{FIN\}$

axm5 : $finite(Send\_Message) \land Send\_Message \neq \emptyset$

axm6 : $last = card(Send\_Message) - 1$

axm7 : $\forall n \cdot n \in dom(Send\_Message) \Rightarrow 0 \mathinner{..} n \subseteq dom(Send\_Message)$

axm8 : $dom(Send\_Message) = 0 \mathinner{..} last$

axm9 : $\forall m \cdot m \in dom(Send\_Message) \land m \neq last \Rightarrow Send\_Message(m) \neq FIN$

axm10 : $Send\_Message(last) = FIN$

axm11 : $\forall n \cdot n \in \mathbb{N} \land n < last \Rightarrow Send\_Message(n) \in Data$

axm12 : $DataTDL \in \mathbb{N}$

axm13 : $DataTDL > 0$

axm14 : $\forall m, x, y \cdot m \in Send\_Message \land m = x \mapsto y \land y = FIN \Rightarrow x = last$

axm15 : $\forall x \cdot x \in dom(Send\_Message) \Rightarrow x \in 0 \mathinner{..} last$

axm16 : $\forall x \cdot x \subseteq \mathbb{N} \land finite(x) \land x \neq \emptyset \Rightarrow max(x) + 1 \notin x$

END

As shown, *Send_Message* represents the message which is supposed to be transferred. Its first packet's id is *0*, and there is an incremental order between the packets' ids. As a result, the last packet which is marked by *FIN* has an id, which is the same as the total number of packets minus one (since the ids start from 0). The last packet's id is kept by constant *last*. Based on the axioms *axm6*, the message is not empty, so the minimum value of *last* is 0.

In the most abstract machine, the system is modelled by three events, *Start_Transferring*, representing the beginning of the transferring process, *Transferred*, representing the accomplishment of the process, and *Error* which represents the occurrence of an unrecoverable error, during this process.

An Event-B Specification of m00
Creation Date: 13Jun2012 @ 06:43:34 PM

MACHINE m00

SEES c00

VARIABLES

$transferred$ {Whole transferred Message}

$Transferred$ {Flag}

$Start\_Transferring$ {Flag}

$Error$ {Flag}

INVARIANTS

inv1 : $transferred \subseteq Send\_Message$

inv2 : $Start\_Transferring \in BOOL$

inv3 : $Transferred \in BOOL$

inv4 : $Error \in BOOL$

inv5 : $time \in \mathbb{N}$

inv6 : $dom(Send\_Message) = 0 \mathinner{..} last$

TIMING

tim1 : $Deadline(Start\_Transferring, Transferred \lor Error, DataTDL)$

EVENTS

Initialisation

begin

act1 : $transferred := \emptyset$

act2 : $Transferred := FALSE$

act3 : $Start\_Transferring := FALSE$

act4 : $Error := FALSE$

end

Event *Start_Transferring* $\widehat{=}$

when

grd1 : $Start\_Transferring = FALSE$

then

act1 : $Start\_Transferring := TRUE$

end

Event *Transferred* $\widehat{=}$

when

grd1 : $Transferred = FALSE$

grd2 : $Start\_Transferring = TRUE$

grd3 : $Error = FALSE$

then

act1 : $Transferred := TRUE$

act2 : $transferred := Send\_Message$

end

Event *Error* $\widehat{=}$

when

grd1 : $Start\_Transferring = TRUE$

grd2 : $Transferred = FALSE$

grd3 : $Error = FALSE$

then

act1 : $Error := TRUE$

end

END

As shown, In this level of abstraction, there is only a deadline, forcing event *Transferred* or event *Error* to happen, within *DataTDL* time-units of event *Start_Transferring*'s occurrence. So if the transferring process has not finished by its deadline, an unrecoverable error will occur in the system.

Besides, since events are not parametrized in this level, the order is modelled by using boolean flags.

9.3.2 The Second Level of Abstraction

In the first refinement, the gradual nature of the transferring process is introduced, in the model. After initiating the transferring process, concrete iterative event *Transferring*, models the process of transferring a message, packet by packet. So the last occurrence of the iterative event, which represents the successful transferring of the last packet refines the abstract event *Transferred*.

9.3.2.1 Refining an Event by Single Occurrence of an Iterative Event

Based on the refinement consistency verification mechanism of Event-B, if an abstract event A, is refined by the last step (or the first) of a sequence of concrete sub-steps, and all of those sub-steps are modelled by a concrete iterative event B , two refinements are required to replace A, by B. As shown in Figure 9.2, in the first refinement, the last step will be modelled by a separate event.

In the first refinement, the guards and the actions of A, will be replaced by the guards and the actions of B, for the corresponding occurrence. The consistency of this refinement can be proved in the usual way using gluing invariants. The second refinement, represented in Figure 9.3(c), is the refinement where events A ($LastB$) and B are merged.

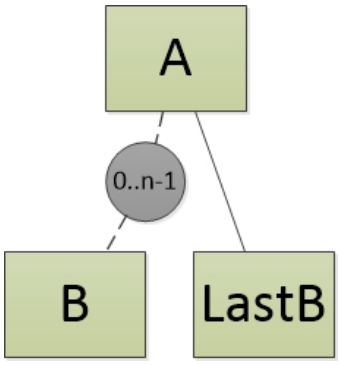
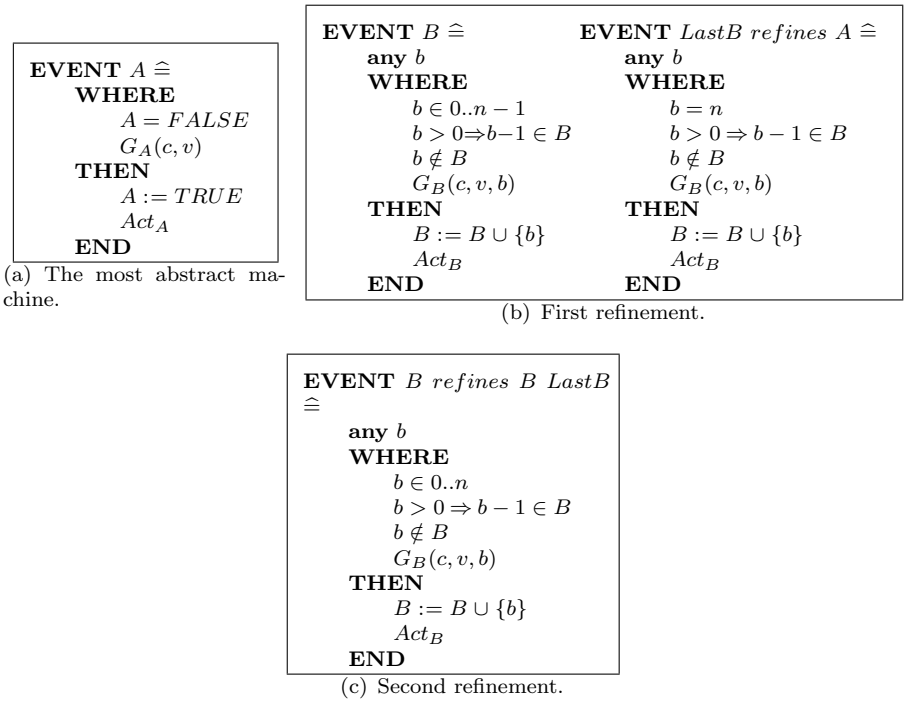


Figure 9.2: Refining an abstract event by the last step of a sequence of concrete sub-steps.



(a) The most abstract machine.

(b) First refinement.

(c) Second refinement.

Figure 9.3: How an abstract event A has been refined by the first occurrence of an iterative concrete event B.

If in the example of Figure 9.3, we refine the abstract machine, represented in 9.3(a), by the concrete machine of Figure 9.3(c), without using the intermediate refinement of 9.3(b), the refinement consistency proof obligations cannot be discharged. This is because the guards of the abstract event do not hold for all the occurrences of B. [1]

The same process has to be performed, in order to replace the abstract *Transferred* event, with the last occurrence of concrete event *Transferring*. The concrete timing property will be as follows:

$$Deadline(Start\_Transferring,\ TransferringLast\ \vee\ Error,\ DataTDL)\,. \quad (9.1)$$

This refinement highlights one of the disadvantages of assuming a fixed approach to encoding event sequencing, in the semantics of timing properties. As shown in the gear controller case-study (Chapter 7), the assumed approach, is practical in most cases. But in a refinement such as this, where an unparametrized event's sequencing has to be enforced by a set ($last \in Trasnferring$), instead of a boolean flag, the semantics is not flexible enough. But this problem can be solved by letting the modeller specify the predicate that determines the occurrence state of a trigger or a response event. As a future work, the approach can be improved by adding more flexibility such as this.

[1] This is just the technicality of the refinement in Event-B

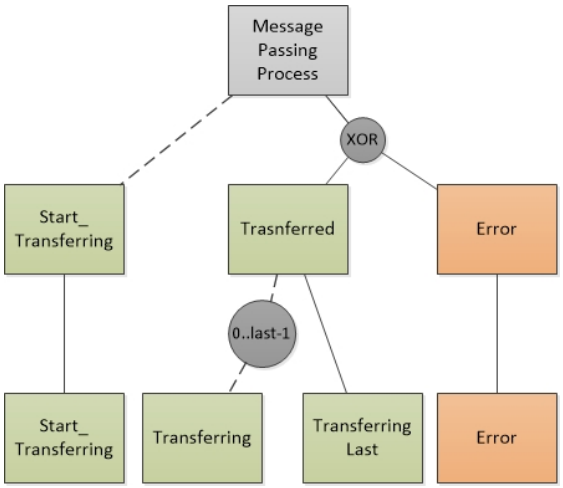


Figure 9.4: The refinement diagram of the first refinement.

So in this level of abstraction, the guard of deadline (9.1), will be as follows:

$$\begin{aligned} &Start\_Trasnfering \ = TRUE \\ &\quad \wedge last \ \notin Trasnferring \Rightarrow time + tick \ \leq tStart\_Trasnfering + DataTDL\,, \end{aligned} \tag{9.2}$$

Since the occurrence of *TransferringLast*, has to be determined by checking whether *last* has been added to the *Trasnferring* set or not.

9.3.3 The Third Level of Abstraction

This refinement aims at replacing the abstract deadline, by a concrete iterative deadline, constraining each packet transferring process. Beside, as mentioned in previous section, abstract event *TransferringLast*, will be replaced by the occurrence of event *Transferring* for parameter *last*. Accordingly, the concrete timing properties, replacing the abstract deadline, are as follows:

$$\begin{aligned} &Deadline(Start\_Transferring,\ Transferring(0) \\ &\quad \vee\ \ Error,\ \ PacketTDL)\ \ replaces\ \ 9.1\,, \end{aligned} \tag{9.3a}$$

$$\begin{aligned} &\forall x \cdot x < last \\ &\qquad \mid\ Deadline(Transferring(x), Transferring(x+1) \\ &\qquad\qquad \vee\ \ Error, PacketTDL)\ \ replaces\ \ 9.1\,. \end{aligned} \tag{9.3b}$$

Deadline (9.3a) is a timing property between an unparametrized event, and the first occurrence of a parametrized one. This deadline specifies the maximum latency between the beginning of the process, and the accomplishment of the first packet's transferring.

Deadline (9.3b) is an iterative timing property, specifying the maximum latency between each packet's transferring and the next one. As mentioned before, *last* represents the last packet's id.

As explained in Section 8.3.2, in order to replace an abstract deadline with an iterative concrete deadline, we need to prove that the overall effect of the iterative deadline, satisfies its abstract. As a result, we need invariants which specify the overall effect, based on the order of sending packets (a packet with the lower id will be sent earlier than a packet with a higher id), and the concrete deadlines. These invariants are as follows:

$$Start\_Transferring = TRUE \wedge 0 \notin Transferring \wedge Error = FALSE \Rightarrow time \leq Start\_TransferringT + PacketTDL\,, \tag{9.4a}$$

$$0 \in Transferring \wedge Error = FALSE \Rightarrow TransferringT(0) \leq Start\_TransferringT + PacketTDL\,, \tag{9.4b}$$

$$\forall x \cdot x < last \wedge x \in Transferring \wedge x + 1 \notin Transferring \wedge Error = FALSE \Rightarrow time \leq TransferringT(x) + PacketTDL\,, \tag{9.4c}$$

$$\forall x \cdot x \in Transferring \wedge Error = FALSE \Rightarrow TransferringT(x) \leq Start\_TransferringT + (x+1) * PacketTDL\,. \tag{9.4d}$$

Invariant (9.4a) is required to prove invariant (9.4b), which specifies the relation between the first transfer's occurrence-time, and the beginning of the process. Invariant (9.4c) (based on deadline(9.3b)) and invariant (9.4b), are required to prove invariant (9.4d). Invariant (9.4d) is the key invariant which connect the occurrence time of each packet's transferring, to the initiation of the message transferring process.

Based on invariant (9.4d) all the packets should be transferred within *PacketTDL*(last+1)* time-units of the beginning of the process, if no error has occurred.

$$DataTDL \geq PacketTDL * (last + 1). \tag{9.5}$$

So, if the iterative concrete deadline's duration, satisfies property (9.5), then the concrete timing properties will satisfy their abstract deadline.

9.3.4 The Forth Level of Abstraction

In this level, the *Sending* event as the pre-step of each packet transferring is added to the model. Event *Sending* represents the process of sending a packet by the sender.

Besides, as shown in Figure 9.5, sending the first packet, which is equivalent to the occurrence of the *Sending* event for 0, refines the *Start_Transferring* event's occurrence. Similar to the refinement process explained in Section 9.3.2.1, In the next refinement sending the 0 packet, will be refined by the *Sending* event.

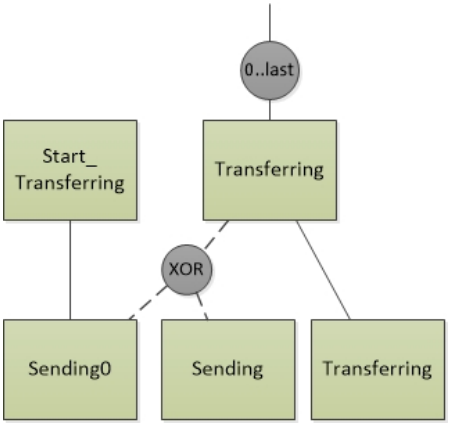


Figure 9.5: Refinement diagram of the forth refinement

Based on the concrete order, sending a packet triggers its *Trasnferring* event. The effect of this change on the abstract deadlines are as follows:

$$\forall x \cdot x \in \mathbb{N} \\ \mid Deadline(Sending(x), Transferring(x) \\ \vee Error, PacketTDL)\ replaces(9.3b)\,, \tag{9.6a}$$

$$\forall x \cdot x < last \\ \mid Deadline(Transferring(x), Sending(x+1) \vee Error, 0)\ replaces(9.3b)\,. \tag{9.6b}$$

Since event *Sending0* has the same guards and actions as the occurrence of the *Sending* event for the first packet, having *Sending(x)* as a response, where x can be 0, will includes the deadline between event *Sending*0, and the *Transferring* event.

Deadline (9.6b) enforces a zero latency between accomplishment of a packet transferring, and sending the next packet. As a result, based on the refinement pattern introduced in Section 8.3.1, the abstract iterative deadline between each packet transferring, and the next one (deadline (9.3b)), has been replaced by two sequential iterative concrete deadlines, one between each sending and its corresponding transferring event (deadline (9.6a)), and another between each transferring occurrence and the sending of the next packet (deadline (9.6b)).

9.3.5 The Fifth Level of Abstraction

By this refinement, the resending process is added to the model, represented by the *ResendingProcess* event. This event can only happen, if the *Sending* event had already

happened for a packet, but the *Transferring* event has not. Besides, the *Resending-Process* event cannot iterate more than $MaxResend$ times. Constant *MaxResend* is declared in the context and it is a natural number.

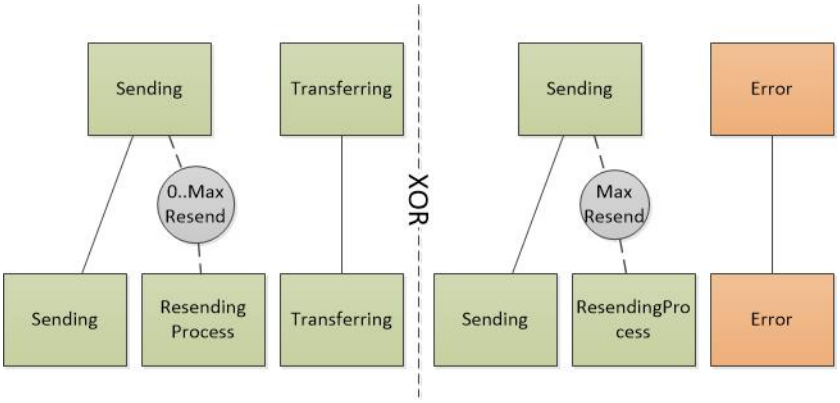


Figure 9.6: The fifth level of abstraction.

As shown in Figure 9.6, after an occurrence of the *Sending* event, either the *Transferring* event's occurrence will follow it or the *ResendingProcess* event's occurrence. The *ResendingProcess* event iterates until either the *Transferring* event happens, or it hits its iteration's upper-bound. After, the last resend, if the *Transferring* does not happen, the *Error* event will happen, representing the occurrence of an unrecoverable error. As a result, either the packet will be transferred successfully by an occurrence of the *Transferring* event, or the system will go to an unrecoverable error state by an occurrence of the *Error* event.

Based on the concrete order, and the refinement pattern explained in Section 8.3.2, the abstract deadline between the *Sending*, the *Transferring*, and the *Error* events, is replaced by deadlines 9.7a, 9.7b, and 9.7c, as follows:

$$\begin{aligned}&\forall x \cdot x \in \mathbb{N} \\ &\quad \mid Deadline(Sending(x \mapsto 0), Transferring(x \\ &\quad \mapsto 0) \vee \ ResendingProcess(x \mapsto 1), ResendingDL)\ replaces(9.6a)\,,\end{aligned} \tag{9.7a}$$

$$\begin{aligned}&\forall x, y \cdot x \in \mathbb{N} \wedge y < MaxResending \\ &\quad \mid \ Deadline(ResendingProcess(x \mapsto y), Transferring(x \mapsto y) \\ &\qquad \vee \ ResendingProcess(x \mapsto y+1), ResendingDL)\ replaces(9.6a)\,,\end{aligned} \tag{9.7b}$$

$$\begin{aligned}&\forall x \cdot x \in \mathbb{N} \\ &\quad \mid Deadline(ResendingProcess(x \mapsto MaxResending), \\ &\quad Transferring(x \mapsto MaxResending) \\ &\qquad \vee Error, ResendingDL)\ replaces(9.6a)\,,\end{aligned} \tag{9.7c}$$

$$\begin{aligned}&\forall x, y \cdot x < last \wedge y \in \mathbb{N} \\ &\quad \mid \ Deadline(Transferring(x \mapsto y), Sending(x+1 \mapsto 0), 0)\ replaces(9.6b)\,.\end{aligned} \tag{9.7d}$$

Deadline (9.7a) is between event *Sending* as the trigger, and the first occurrence of the *ResendingProcess* event, or the corresponding occurrence of the *Transferring* event, as the response events. Deadline (9.7b) is an iterative deadline between each occurrence of the *ResendingProcess*, and the corresponding occurrence of the *Transferring* event, or the next occurrence of the *ResendingProcess*, for the same packet. The (9.7c) deadline is between the last occurrence of the *ResendingProcess* as the trigger event, and either occurrences of the *Error* event, or the *Transferring* event, as the deadline's responses. These deadline have replaced the abstract deadline (9.6a).

A new parameter (represented by y in some of the deadlines) exists in some of the concrete timing properties, used to show which sending attempt, a timing property is about. Concrete deadline (9.7d) refines abstract deadline (9.6b), by including the new parameter.

Similar to what has been explained, in the refinement pattern 8.3.2, the timing property refinement, presented in this section, is consistent if:

$$PacketTDL \geq ResendingDL * (MaxResending + 1) \tag{9.8}$$

So the overall deadline duration, resulted by the concrete deadlines, has to be less than or equal to the abstract deadline's duration.

But these are not all of the concrete timing properties. In this level, some delays, constraining the occurrence of the *ResendingProcess*, and the *Error* events, have been introduced too. These delays give enough time to a packet, and its acknowledgement, to travel through the channel. So the resending only happens, if the sender believes, the transferring process for the previously sent packet was not successful (based on time). For the last resending attempt, the delay prevents the occurrence of an unrecoverable error, if there is still time for the packet to be transferred. These delays are as follows:

$$\forall x \cdot x \in \mathbb{N} \mid Delay(ResendingProcess(x \mapsto MaxResend), Error, ResendingDL)\,, \tag{9.9a}$$

$$\forall x, y \cdot x \in \mathbb{N} \wedge y < last \mid Delay(Sending(x \mapsto y), ResendingProcess(x \mapsto y+1), ResendingDL)\,, \tag{9.9b}$$

$$\forall x, y \cdot x \in \mathbb{N} \wedge y < last \mid Delay(ResendingProcess(x \mapsto y), ResendingProcess(x \mapsto y+1), ResendingDL)\,. \tag{9.9c}$$

Delay (9.9a) is between the last resending attempt, and the error occurrence. Delays (9.9b) and (9.9c) are on the *ResendingProcess* event, the *Sending* event and the first occurrence of the *ResendingProcess* event, and the other between the previous resend and the next one.

9.3.6 The Sixth and Seventh Levels of Abstraction

In these two refinements, first the *DataLost* event, representing the loss of a packet, and *ACKLost* event, representing the loss of an acknowledgement, in the channel, are added to the model. By having these two events, in the next refinement, the abstract event *Transferring* is refined by the *RecevingACK* event, representing the arrival of an acknowledgement in the sender.

Besides, the receiver's events, *Receiving* and *Rereceiving* are added. The *Receiving* event represents the arrival of a packet in the receiver, and the *Rereceiving* event represents the arrival of a packet in the receiver, which has been already received. By having these receiver's events, its error event can be added to the model too. As explained in the requirements section (9.1.4), the receiver will goes to an unrecoverable error state, if the last packet has not been received, and the receiver does not hear from the sender, within *ReceivingDL* time-units of the previous data arrival.

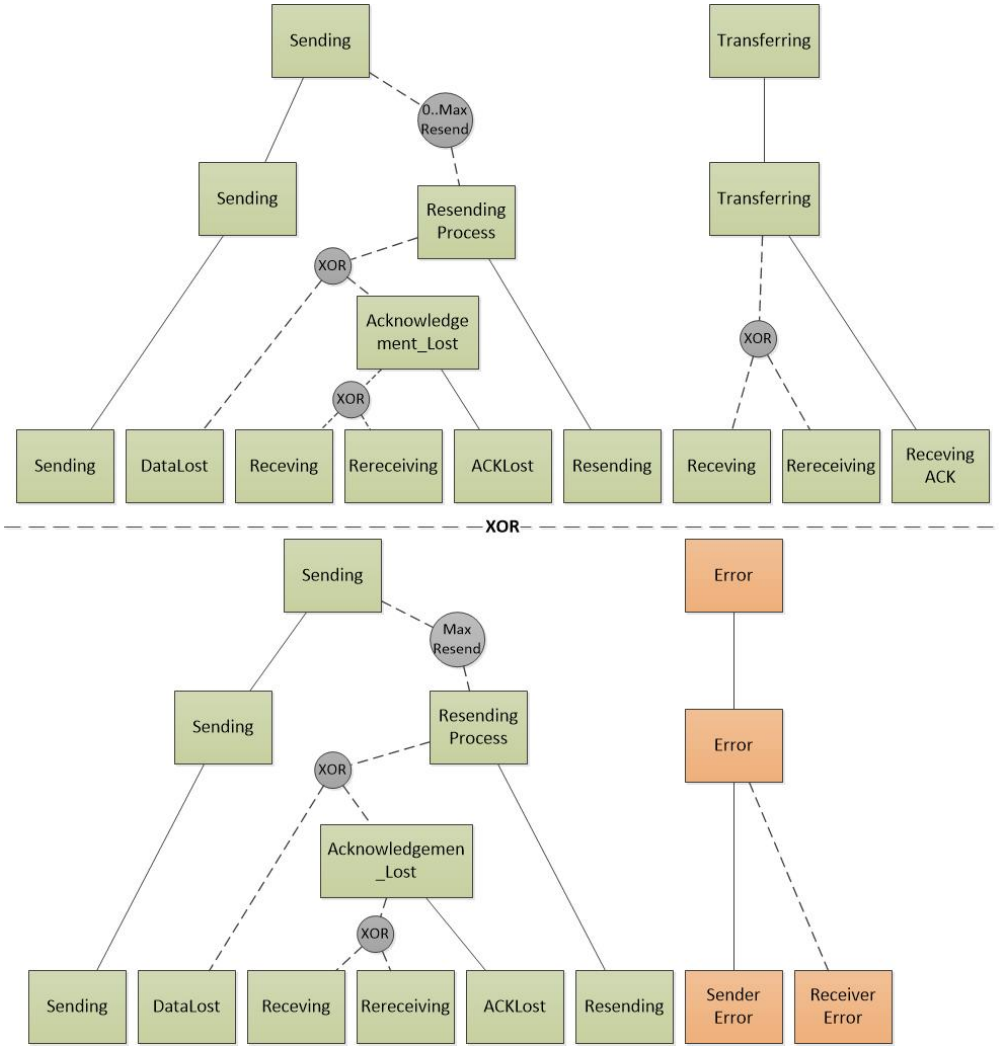


Figure 9.7: The refinement diagram of the fifth and sixth refinements

As shown in Figure 9.7 after $MaxResend + 1$ times, transferring attempts of a packet, the sender will give up, and the receiver will eventually go to an error state. In the diagram there is an intermediate event *Acknowledgement_Lost*, which does not exist in the Event-B model presented in Appendixes (Section A.4). It is just added to the diagram, in order to improve its readability.

Since abstract event *ResendingProcess*, has been refined by concrete event *Resending* (the abstract event's name, has been changed), it will be replaced by it, in all the abstract timing properties.

Besides, by adding these concert events, it is now possible to specify the timing properties of the channel and the receiver. Their timing properties does not replace any abstract timing property, since they are the properties of behaviours, hidden in the abstraction. In the abstraction, the transferring process was modelled from the sender perspective. As a result all of the abstract timing properties, belongs to the sender. On the other hand, the newly added timing properties in these two refinements, express the properties of the receiver and the channel. These concrete timing properties are as follows:

The channel's timing properties

$$\begin{aligned} &\forall x \cdot x \in \mathbb{N} \\ &\quad \mid Deadline(Sending(x \mapsto 0),\ Receiving(x \mapsto 0) \\ &\qquad \vee DataLost(x \mapsto 0), ChannelDL)\,, \end{aligned} \tag{9.10a}$$

$$\begin{aligned} &\forall x, y \cdot x \in \mathbb{N} \wedge y \in \mathbb{N} \mid Deadline(Resending(x \mapsto y), Receiving(x \mapsto y) \\ &\qquad \vee Rereceiving(x \mapsto y) \vee DataLost(x \mapsto y), ChannelDL)\,, \end{aligned} \tag{9.10b}$$

$$\begin{aligned} &\forall x, y \cdot x \in \mathbb{N} \wedge y \in \mathbb{N} \mid Deadline(Receiving(x \mapsto y),\ ReceivingACK(x \mapsto y) \\ &\qquad \vee ACKLost(x \mapsto y), ChannelDL)\,, \end{aligned} \tag{9.10c}$$

$$\begin{aligned} &\forall x, y \cdot x \in \mathbb{N} \wedge y \in \mathbb{N} \mid Deadline(Rereceiving(x \mapsto y),\ ReceivingACK(x \mapsto y) \\ &\qquad \vee ACKLost(x \mapsto y), ChannelDL)\,. \end{aligned} \tag{9.10d}$$

The receiver's timing properties

$$\begin{aligned} &\forall x, y_1, y_2, z \cdot x < last \wedge z > 0 \\ &\quad \mid Deadline(Receiving(x \mapsto y_1),\ Rereceiving(x \mapsto y_1 + z) \\ &\qquad \vee Receiving(x + 1 \mapsto y_2), ReceivingDL)\,, \end{aligned} \tag{9.11a}$$

$$\begin{aligned} &\forall x, y_1, y_2, z \cdot x < last \wedge z > 0 \\ &\quad \mid Deadline(Rereceiving(x \mapsto y_1),\ Rereceiving(x \mapsto y_1 + z) \\ &\qquad \vee Receiving(x + 1 \mapsto y_2), ReceivingDL)\,, \end{aligned} \tag{9.11b}$$

$$\forall x, y \cdot x < last \mid \ Delay(Receiving(x \mapsto y), Error, ReceivingDL)\,. \tag{9.11c}$$

Deadlines (9.10a), (9.10b), (9.10c), and (9.10d), specify the maximum duration required for a data packet, or an acknowledgement packet, to travel through the channel. So after *ChannelDL* time-units, of sending a packet, it has either arrived to its destination, or has been lost.

Deadlines (9.11a)and (9.11b) specify the maximum duration, the receiver waits to hear from the sender, before it goes to an error state. At last, delay (9.11c) specified the minimum duration, the receiver waits, before assuming a faulty connection, and going to an unrecoverable error state.

There is no mechanism to inform the sender about a lost in the channel. As a result, the sender activates its packet-loss's strategies, based on time. Similarly, there is no mechanism to inform the receiver about a faulty connection, either caused by an error in the sender, or the channel being out of order. So, the receiver decides about going to an unrecoverable error state, based on time too.

To do this efficiently and effectively, there should be some relations between the timing properties of these three components. These relations are as follows:

$$ReceiverDL \geq MaxResend * ResendingDL + 2 * ChannelDL\,, \tag{9.12a}$$

$$ResendingDL > 2 * ChannelDL\,. \tag{9.12b}$$

Based on (9.12a) the maximum duration, the receiver waits before going to an unrecoverable error state is longer than sequence of the following durations:

1. the duration required for the acknowledgement to get to the sender (*ChannelDL*),
2. the duration required for the sender, to try transferring a packet $MaxResedning$ times (MaxResend * ResendingDL),
3. the maximum duration required for the last sending attempt of a packet, to get to the receiver (*ChannelDL*).

Based on (9.12b) the required time for a packet and its acknowledgement to travel through the channel should be less than the duration the sender waits before resending that packet. So, the sender just resends a packet if it has already been lost. The satisfaction of this property, by this model, has been proved as an invariant, in the next refinement, explained in the following section.

9.3.7 The Eighth Levels of Abstraction

To prevent an overly complex machine, in the previous refinement, how the timing properties of the channel, the sender, and the receiver, synchronize them has been verified

in this refinement. In this level, the aim is to prove that by the time, the sender resends a packet, its previous attempt has been already failed, either by losing the packet itself before getting to the receiver, or by the loss of its acknowledgement. This has been done by adding an invariant. This invariant is not a gluing invariant, so it is not required to prove the consistency of a refinement.

As mention in Section 9.3.6, based on the relation of the timing properties' durations, the sender can recognize whether a resend is required or not. The following invariant encodes this property:

$$\begin{aligned}\forall x, y \cdot x \mapsto y \in Resending \Rightarrow\ & (x \mapsto y-1 \in Receiving \cup Rereceiving \\ & \wedge\ x \mapsto y-1 \in ACKLost) \vee\ (x \mapsto y-1 \in DataLost)\end{aligned} \tag{9.13}$$

Invariant (9.13) can be proved based on the timing properties of the system. But first, we need some other invariants which specify the relation of the events' occurrence-times, in different components. These invariants are as follow:

$$\forall x \cdot x \mapsto 0 \in Sending \wedge x \mapsto 0 \in Receiving \Rightarrow\ ReceivingT(x \mapsto 0) \leq SendingT(x \mapsto 0) + ChannelDL\,, \tag{9.14a}$$

$$\forall x, y \cdot x \mapsto y \in Resending \wedge x \mapsto y \in Receiving \Rightarrow\ ReceivingT(x \mapsto y) \leq ResendingT(x \mapsto y) + ChannelDL\,, \tag{9.14b}$$

$$\forall x, y \cdot x \mapsto y \in Rereceiving \Rightarrow x \in dom(Receiving)\,, \tag{9.14c}$$

$$\forall x, y \cdot x \mapsto y \in Resending \wedge x \mapsto y \in Rereceiving \Rightarrow ReceivingT(x \mapsto y) \leq ResendingT(x \mapsto y) + ChannelDL\,, \tag{9.14d}$$

$$\forall x, y \cdot x \mapsto y \in Rereceiving \Rightarrow\ x \mapsto y \in Resending\,. \tag{9.14e}$$

As shown, most of these invariants are about the channel's deadlines. Based on the channel deadlines, and the fact that the acknowledgement will be sent at the same time as the packet is received, it is possible to prove that, if the sender waits more than *2*ChannelDL*, and does not receive any acknowledgement, the only possibility is the loss of the packet, or its acknowledgement. These invariants mostly specify how long it takes for a packet to travel through the channel. As a result if it has not arrived by that time, the only possibility is that it has been lost.

So far, all of the details of the system specification have been added to the model. To evaluate the practicality of the introduced decomposition approach in Section 5, the following section goes through the process of decomposing the most concrete model of this case-study.

9.3.8 The Ninth Levels of Abstraction and Decomposition

As explained in Section 5, in order to decompose a timed Event-B model, the ordering sets or flags, and the occurrence time variables of the shared events, need to be replicated, and the *time* variable needs to be replicated for each component. Besides, in the timing properties of each component, its corresponding replicate of the time variable should be used. This is what has been done in the ninth level of abstraction. Three components are assumed for this system, the sender, the receiver and the channel. As a result, there are three copies of the *time* variable. In this decomposition, the shared events are as follows:

- Between the sender and the channel:

  Sending

  Resending

  ReceivingACK

  Tick_Tock

- Between the receiver and the channel:

  Receiving

  Rereceiving

  ReceivingACK

  Tick_Tock

- Between the Sender and the Receiver:

  Tick_Tock

The *Tick_Tock* event is shared between all three components. Other than that, the sender and receiver are not connected directly, and all the communications are done through the channel. After decomposition, the timing properties will be decomposed too, so in the *Tick_Tock* event of each component, only the deadline guards of that component will appear.

9.4 Achivements

In this case-study, a message passing protocol between a sender and a receiver, through a channel have been modelled. The case-study has demonstrated the capability of the approach to model interconnected iterative processes. It has done by starting from an abstract machine, in which hides the iterations and reflects the overall effects, and

properties of them (including timing properties), and evolving it to include the iterative behaviours of the system ,and their properties by using the refinement patterns, introduced in Sections 4.3 and 8.3.

Machine	Number of Generate PO	Automatically Proved	Automatically Proved%
m00	8	8	100
m01	44	39	88
m02	26	23	88
m03	70	66	94
m04	66	56	84
m05	8	8	100
m06	82	74	90
m07	49	38	77
m08	47	45	95
Total	406	362	89

Table 9.1: Number of generated proof obligations for each machine and how they were proved

In this way, the consistency between the overall properties, and the properties of each iteration has been proved. We have presented how the timing properties, and the order of events' occurrences, provide the desirable overall behaviour of the system. The number of generated proof obligations and how they have been proved in each machine have been presented in Table 9.1.

Although the portion of interactively proved POs is not low, but most of those POs have been proved by using the new prover plug-in *MetaProver*. The advantage of the *MetaProver*, is its ability to collect the more relevant hypothesises, in order to discharge a PO. This is caused by the occurrence sets, since the *Atelier B* prover, is not effective enough to prove the invariants, containing sets.

Chapter 10

Timing Properties Plug-in

As mentioned in Section 4.2, based on the semantics of timing properties, several variables, invariants, guards and actions are needed to be added to an Event-B model in order to encode them. Besides, in order to verify the consistency of the timing properties' refinements, as it was discussed in Section 4.3, some gluing invariants are required.

Going through this process manually, is time consuming and fallible. By automation of this process, since the semantics will be implemented automatically, the modeller just needs to know the syntax of timing properties, in order to use the approach.

To achieve this level of practicality, we have extended the Rodin tool-set [8] to support the timing properties based on this work. Since Rodin is an Eclipse-based IDE [5], it can be extended with plug-ins.

In this report we will not go into the details of the Rodin extension process, but there are some useful information available in the Event-B wiki page [7]. In the following, the features of the timing properties plug-in, and how they can facilitate the process of modelling and verifying a real-time system in the Rodin tool-set, will be discussed.

10.1 Timing Plug-in's Features

The plug-in is supporting unparametrized timing properties in this stage. Our goal is to hide the details of encoding timing properties from the modeller. A special interface has been designed for the timing, which offers the following facilities:

- Presenting the existing timing properties of the corresponding machine,
- Providing the delete function for the timing properties,
- Editing the existing timing properties (changing the trigger or response events, or changing the timing properties duration),

- Adding a new timing property,
- Sorting the events alphabetically in order to improve the search experience when the modeller wants to choose the trigger and response events.

In the following, some of these facilities will be explained in more details.

10.1.1 Adding a New Timing Property

In order to add a new timing property, the modeller has to choose the type of the property from a list (deadline, delay or expiry), a trigger event from a list of all the existing events of the corresponding machine, except the *Tick_Tock* and *INITIALIZATION* events, a response event (or several in the case of deadline) for the similar list, and a duration from a list of all the integer constants, visible to the machine (declared in the contexts seen by the machine). Then the tool will generate all the required time related variables, typing invariants, initialization actions, timing guards, occurrence recording actions, and the *Tick_Tock* event.

In addition, the required time related gluing invariants, in order to discharge the refinement consistency proof obligations, will be generated by the tool. But, some compromises have been made in terms of generated gluing invariants. Since, during the process of adding a timing property, the the overall timing paradigm of the corresponding machine, may not be available, we are generating the invariants which connect the occurrence time of a response event to its trigger event based on the timing property, whether they are required based on the refinement, or not.

We do not believe that, because they are making the model more complex, having them is a disadvantage. Those invariants, specify the timing properties of the system, and discharging their corresponding POs, verify the satisfaction of those properties by the model. Besides, their corresponding POs can be discharged by the automatic prover, so their existence does not cause any inconvenience.

As explained in Section 4.3, if a timing property is refined by a sequence of several sub-timing properties, some gluing invariants will be required to specify the correlation between the occurrence times of the intermediate trigger events and the occurrence time of the abstract trigger event, based on those concrete sequential sub-timing properties.

Similar to what has been explained in refinement pattern 4.3.1, if in a trigger-response pattern, the response event is refined by several sequential sub-responses, in order to prove the consistency of the refinement, the relation between the occurrence-time of the last sub-response, in the concrete sequence of responses, and the abstract trigger event has to be specified. This is done, by specifying the relation between the occurrence-times of each sub-response, and its trigger event (for the first sub-response, it is the abstract

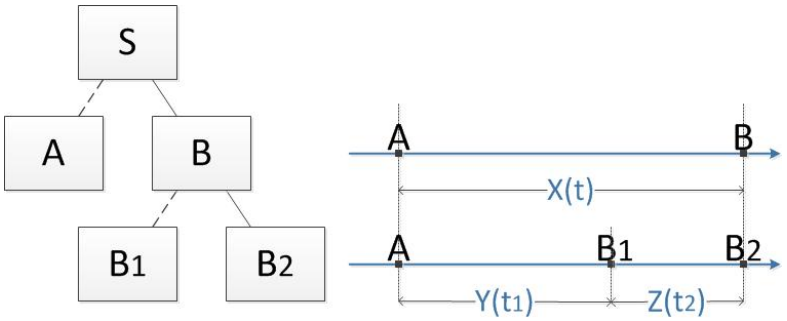


Figure 10.1: Refining a timing property of type X, to a sequence of two concrete sub-timing properties of type Y and Z.

trigger event, and for the rest, it is the previous sub-response, in the sequence of concrete sub-responses), based on their timing property, in terms of an invariant.

For the refinement pattern of Figure 10.1, the relation between the occurrence-times of A and B_1, will be specified based on timing property Y, and the relation between the occurrence-times of B_1 and B_2, will be specified based on Z.

These type of invariant will be generated for all the timing properties of a model, by the timing plug-in, whether they are required to prove the consistency of a timing refinement, or not. As a result despite the timing properties' semantics, introduced in Section 4.2, where just the occurrence-time of a trigger event is recorded, our plug-in will generate the required component to recorded the occurrence-time of each response event, too.

By considering the amount of effort required to generate these invariants by hand, this change seems to be an acceptable compromise. Besides, it does not affect anything, which has been explained so far.

$$Deadline(A, B, t)\,, \tag{10.1a}$$

$$Delay(A, B, t)\,, \tag{10.1b}$$

$$Expiry(A, B, t)\,. \tag{10.1c}$$

The generated invariants by the plug-in, for a deadline such as (10.1a), are as follows:

$$A = TRUE \wedge B = TRUE \Rightarrow tB \leq tA + t\,, \tag{10.2a}$$

$$A = TRUE \wedge B = FALSE \Rightarrow time \leq tA + t\,. \tag{10.2b}$$

As explained in Section 4.3.1, invariant (10.2b) is required to prove invariant (10.2a) for event B, since the timing property has guarded the $Tick\_Tock$ event, and the required information about time to discharge the corresponding POs of invariant (10.2a) is not available in the guards of B.

Since, delays and expiries guard the corresponding response event, we just need the invariant which connect the occurrence time of the response event to the trigger one, based on their timing property, and there is no need for any other timing invariant to discharge its POs. As a result, the generated invariants to specify the correlation between the occurrence time of a trigger event and its response event's occurrence time for a delay and an expiry such as (10.1b) and (10.1c) are as follow:

$$A = TRUE \wedge B = TRUE \Rightarrow tB \geq tA + t, \tag{10.3}$$

$$A = TRUE \wedge B = TRUE \Rightarrow tB \leq tA + t. \tag{10.4}$$

Invariant (10.3) will be generated for a generic delay (10.1b), and invariant (10.4) for expiry (10.1c).

The other invariant that may be generated by the timing plug-in, specify the correlation between occurrence time of an alternative concrete trigger event and its abstract one.

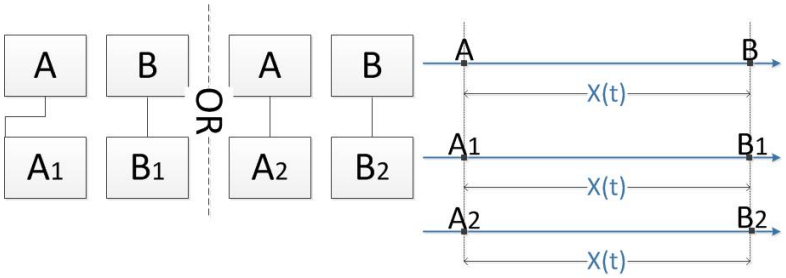


Figure 10.2: Refining a trigger-response pattern, by two alternative trigger-responses

Similar to what has been explained in Section 4.3.4, if a timing property of type X is refined by two alternative timing properties of the same type as shown in Figure 10.2, the following invariant is required to connect the abstract trigger event occurrence time and the concrete occurrence times of the alternative trigger events:

$$tA = tA_1 \wedge tA = tA_2 \tag{10.5}$$

Invariant (10.5) will be generated, if the trigger event of the added timing property, refines an event, which has a timing property and a different name.

The gear controller case-study, has been remodelled, by using the plug-in, and no time related invariant was required to be declared by the modeller in order to discharge the refinement consistency POs. Since the case-study covers a wide range of timing properties and their refinement, it can be claimed that the tool at least supports all the unparametrized refinement patterns, introduced in Section 4.3. In the next section the advantage and disadvantage of the timing plug-in will be discussed in more details.

10.1.2 Advantages & Disadvantages of the Timing Plug-in

The biggest advantage of having a tool support for the approach, is the improvement of practicality. By using the plug-in, adding a timing property to an Event-B model, removing one or editing it, can be done in a matter of a minute. Whereas, if it is supposed to be done manually, it requires a lot of effort. Other than the time, the modeller has to spend in order to learn the semantics of the timing properties and the require invariants to discharge the refinement consistency POs, encoding timing properties is a hassle

Based on our experiences, a timing property can be added to a simple Event-B model with a pair of trigger-response events, in two and half minutes by hand if the modeller is well experienced with the approach and Event-B, whereas it takes less than ten seconds to do it by using the plug-in. Considering, the simplicity of the machine and consequently its timing properties, this figure will be more impressive for a complex model, with complex timing properties (deadlines with several alternative responses). Besides, it should be considered that in this example, there is no refinement. Adding the abstract timing properties to a concrete machine and declaring the required gluing invariants is a complex and error prone process by its own, which can be done in a matter of a second by the plug-in.

The gear controller case-study was one of the early case-studies we have done during this work. For this report, based on the mature semantics (improved gradually, by applying them on different examples), it had to be redo. Since I was completely familiar with the specification of the system, I just needed to think about the modelling process and decide about different levels of abstractions and their timing properties. First, I added the timing properties by hand, and it took me about two weeks to do the whole thing. Then I removed all the timing properties and their related variables, guards, actions, invariants and events from the model, and this time we added the timing properties by using the plug-in. The whole process of adding the timing properties and proving their consistency, just took me a day.

Even by considering the required time to model the untimed model of the system in the first attempt, which was about a week, there is an impressive difference between the required time with and without the plug-in. Besides, it should not be forgotten that as the designer of the approach I am well familiar and comfortable with it, so I can add timing properties by hand much faster than a typical Event-B modeller. As a result, it can be claimed that these results will be much better for a typical Event-B modeller.

But there are some issues too. First of all, after adding timing properties to an abstract machine and its refinement, if a new timing property is added to the abstract machine, or an existing one is changed, the concrete machine will not be updated accordingly. Besides, the unparametrized timing properties are not supported yet. The other issue which may be considered as a problem, is the number of generated invariants. As

explained in Section 10.1.1, since the overall picture is not available in the process of adding a timing property, some invariants will be generated which are not necessarily required. But, since they all can be discharged by the automatic prover, the complexity they add to the model, is not really an issue.

For the tenth levels of abstraction, in which the gear controller case-study has been modelled (minus the level required for the decomposition), the model generated by hand has 1100 POs, from which 1052 of them have been discharged automatically (95% were automatically dischahrged). In the model which the timings have been added by using the plug-in, 2257 proof POs have been generated, from which 2202 (97% of POs have been discharged automatically). As these numbers show, although the number of generated proof obligations is much higher in the case of using the plug-in, but the ratio of automatically discharged POs is still better when the plug-in is used, since all the corresponding POs of those unneccessary invariants (which do not exist in the manually generated model), have been discharged automatically.

The automatic provers have some limitations. For example, in complex models, with so many events and invariants, the automatic provers cannot find the relevant hypotheses to discharge a PO, before their time-out. In the last machine, because of the complexity of the model (consist of more than 50 events) and its timing properties (more than 50 timing properties) the automatic provers were unable to discharge some of the POs. The only thing modeller can do, is to open the interactive interface, and run the *ML* prover from there. So it is not actually an interactive prove, it is just running an automatic prover without any time-out. Each PO which has not been discharged by the automatic provers, has been discharged in less than 15 seconds in this way.

In the model, where timing properties have been added by hand, since the timing gluing invariants were generated more heuristically, there were less of them. Consequently the model was less complex and the automatic provers were able to discharge the POs of the gluing invariants before the time-out. This kind of problem can be solved by improvement of the automatic provers in the future.

Besides, as explained in Section 4.2, we have proved the consistency of the timing properties semantics. As a result, there is no need to prove the timing properties invariants, in every model. So, another improvement of the tool can be extending the POs generator, to prevent the consistency POs of those invariant to be generated, based on the assumption that they are consistent by construct. By this improvement, number of POs of a timed Event-B will be reduced considerable, and there will be less time related POs which modeller has to discharge interactively.

In this chapter, the features of the plug-in developed to support our approach have been discussed and how they improve the modelling experience in Event-B, has been explained. The down-sides of using the tool have been mentioned and a comparison of the gear controller case-study (Chapter 7) modelling experience, with and without the

plug-in has been presented. The developed model by using the plug-in is available in the Appendix (Sections A.2 and A.3).

Chapter 11

Conclusions

This work has been based on Event-B formal modelling and verification language [13]. We have extended the language by several definable discrete timing properties and demonstrated their usefulness and practicality with two case studies.

In Chapter 4, The Event-B syntax was extended by three discrete unparametrized timing properties, based on the trigger-response pattern, and how they are encoded by using the standard constructs of Event-B was discussed. We believe the main contribution of this work is the introduced semantics for timing properties, which supports refinement, decomposition, and mechanized process of extending an untimed Event-B model by timing properties. Besides, in this work, several types of timing properties have been covered. By benefiting from the refinement feature, it is possible to verify the consistency of timing properties in different levels of abstraction, and decomposition helps the modeller to independently refine the timing properties of a component in a large system.

In Chapter 5, our approach to decomposed a timed Event-B model was explained. Based on this approach, it is possible to extend and refine the timing properties of each decomposed machine, independently. Besides, time progresses synchronously in the decomposed machines.

In Chapter 6, we investigated the timing properties' effects on the enableness of response events. The aim is to show that by adding timing properties, response events still have the chance to occur, and the progress of time will not be disabled indefinitely. In most cases, having some simple relations between the durations of timing properties, guarantees the satisfaction of those properties, but proving that the eventual enabling of time progressing event can be challenging.

In Chapter 7, the gear controller case study was presented. This is a reasonably complex and large system, which benefits from different means of synchronization, mainly time.

Our aim was to demonstrate, how timing refinement can be beneficial during the modelling process of a complex real-time system. As presented, most of the proof obligations were discharged automatically. Although, some of the timing properties refinements' POs needs modeller interactions to be discharge, we believe the prover improvement can solve this issue.

In Chapter 8, The Event-B syntax was extended by three discrete parametrized timing properties, based on the trigger-response pattern, and similar to the unparametrized ones, how they are encoded was discussed. Besides, how the introduced refinement patterns, and the decomposition process of unparametrized timing properties can be applied on parametrized ones, were explained.

In Chapter 9 another case study was discussed, in order to demonstrate the practicality of our approach, for parametrized events. Similar to the gear controlling system, the practicality of the timing refinement and decomposition were the focus of the chapter.

In Chapter 10, the features of the timing extension of the Rodin tool-set, developed to support our approach, have been discussed. The chapter aimed to demonstrate, how a tool support can make the approach even more piratical, in terms of the required effort to add timing properties to an untimed Event-B model.

11.1 Related Work

Many studies have been dedicated to formalize and verify timing properties of real-time systems. Delay, deadline and expiry can be seen in many of those works, sometimes with different names.

In real-time calculus TCCS of Wang [116] there is a delay construct $\epsilon(d) \cdot P$, which forces the model to wait for d time-units and then behave as process P and time cannot proceed if d time-units have passed and process P has yet to happen. The similar mechanism has been used in Timed Modal Specification of Cerans et al. in [45] to model maximal progress assumptions, where there is a must modality which enforces the maximum delay to a model.

Delay in TCCS [116], and maximal progress in Timed Modal Specification [45], present the same property as deadline in our work. Also, what is called a loose delay in Timed Modal Specification forces the same behaviour as a delay does in our work.

Urgent Events in Evans and Schneider work [60] have been encoded by preventing time proceeding, if an urgent event is eligible to occur. This behaviour of urgent event is the same as response events of a deadline in our work, when the current time is equal to the deadline and none of the responses have occurred.

In Timed CSP [3], time-out presents the same property as expiry does in our work and a delay in Timed CSP causes a similar behaviour to what can be modelled by combining a delay and a deadline in our work.

Modelling time-critical systems by using Event-B has been investigated in several studies. Butler et al. in [41] explained how it is possible to model discrete time in B, by having a natural number variable which represents the current time and an operation which forwards the time. In that study a deadline has been modelled by disabling the time progress, if the current time is equal to the deadline. This work does not investigate different kinds of timing properties and timing property refinement has not been discussed.

Cansell and Rehm in [44] have modelled a message passing algorithm in Event-B by using similar principles, having a natural number variable representing the current time, and an event which forwards the time, and guarded by some sets of activation times. Again in here, other kinds of timing properties have not been mentioned, but more importantly, as explained in Section 4.5, refining a timing property to finer ones based on this approach is a challenging process. Because, in order to do that, some new values should be added to the activation set in the refinement which is not possible without declaring a new activation set. The problem will be specifying the relation between the new activation set and its abstract one. The User has to show in any given time, the minimum of the concrete activation set is less than or equal to the minimum of its abstract set, which is a complex proof to be done.

Bryans et al. in [37] have introduced an approach to keep track of timing boundaries between different events in a model by adding them to a set, and guarding events based on it. In their study, the deadline has not been modelled. Similar to the previous approach, refining the timing property will be an issue because of the set which tracks the timing boundaries.

A more detailed comparison of our approach and the existing works in modelling and verifying time-critical systems, has been presented in Sections 4.5 and 7.4. Section 7.4, demonstrate the differences of our approach and UPPAAL based on the gear controller case-study, which has been modelled in both approaches.

11.2 Future Work

As mentioned in Chapter 10, a plug-in has been developed which let the modeller to specify the unparametrized timing properties of an Event-B machine, based on the introduced syntaxes, then it will encode them based on the semantics of the timing properties. Besides the required gluing invariants to prove the consistency of the timing refinements will be added by the tool automatically.

The tool is in its early stage, and there are many areas to be improved. For example, generating gluing invariants can be improved to be done more heuristically, and an explicit support for the decomposition of timed Event-B models should be added. Besides, the tool has to be extended to support the parametrized timing properties too. As mentioned in Chapter 10, a possible improvement is to extend the proof obligation generator based on the semantics of timing properties. As shown in Section 4.2, we have proved the consistency of the semantics for generic trigger-response event. As a result there is no need to generate the corresponding POs anymore. In this way, number of POs in a timed Event-B model will be reduced, and the user will not be forced to do the interactive proves of timing invariants anymore. Besides, by standardizing the refinement patterns the same thing can be done for them too.

In Section 3.4, we talked about fairness in TLA. Another possible area of improvement is to investigate fairness in timed Event-B models.

In a real-time program, the sensing and the actuating happen periodically, and the durations of their interleaves provide the abstract timing properties of the controller, which is expressed in terms of deadlines and delays in our approach. As a result, whether the refinement of these timing properties by periodic behaviours is a valid one, can be investigated as a part of future works. In this way, it will be possible to have a concrete machine which represents the implementation in more details.

Appendix A

Event-B Models

In this section the Event-B models of the case-studies, discussed in this report, are presented.

A.1 Event-B Model of the Gear Controller Case-study (Manual)

The model is available in the following address:
http://eprints.soton.ac.uk/345075/1.hasCoversheetVersion/GearManual.pdf

A.2 Event-B Model of the Gear Controller Case-study (Plug-in)

The timing properties, added manually in the most concrete machine (*m9*) of the the model presented in Section A.1, have been added in three levels of abstraction (*m9, m10, m11*) by using the plug-in, in order to decrease the complexity of each machine (less timing gluing invariants per machine).

The model is available in the following address:
http://eprints.soton.ac.uk/344946/1.hasCoversheetVersion/GearPlugin.pdf

A.3 Event-B Model of the Gear Controller Case-study (Improved Plug-in)

As mentioned in Chapter 10, the plug-in have been improved to generate the gluing invariants more efficiently. In the following model of gear controller case-study, the timing

properties have been added by the improved plug-in. So the number of generated invariants is closer to the manual model. In this model, just the first 5 levels of abstraction have been included.

The model is available in the following address:
http://eprints.soton.ac.uk/344950/1.hasCoversheetVersion/GearImpPlugin.pdf

A.4 Event-B Model of the Message Passing Case-study (Manual)

The model is available in the following address:
http://eprints.soton.ac.uk/342272/4.hasCoversheetVersion/MessagePassing.pdf

References

[1] *Linear and Branching Structures in the Semantics and Logics of Reactive Systems*, volume 194 of *Lecture Notes in Computer Science*. Springer, 1985.

[2] *Synchronous Programming of Reactive Systems*, volume 1427 of *Lecture Notes in Computer Science*. Springer, 1998.

[3] *Concurrent and Real Time Systems: The CSP Approach (Worldwide Series in Computer Science)*. John Wiley & Sons, September 1999.

[4] TCOZ. http://www.comp.nus.edu.sg/~dongjs/tcoz.html, August 2010.

[5] Eclipse website. http://www.eclipse.org/, February 2012.

[6] Event-B website. http://www.event-b.org/, February 2012.

[7] Rodin developer support wiki. http://wiki.event-b.org/index.php/Rodin_Developer_Support, February 2012.

[8] Rodin website. http://wiki.event-b.org/index.php/Rodin_Platform, February 2012.

[9] Time. http://plato.stanford.edu/entries/time/#3, June 2012.

[10] UPPAAL website. http://www.uppaal.org, March 2012.

[11] Martín Abadi and Leslie Lamport. The Existence of Refinement Mappings. *Theor. Comput. Sci.*, 82(2):253–284, 1991.

[12] Martín Abadi and Leslie Lamport. An old-fashined recipe for real-time. *ACM Trans. Program. Lang. Syst.*, 16(5):1543–1571, 1994.

[13] Jean-Raymond Abrial. *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, New York, NY, USA, 1st edition, 2010.

[14] 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. Transf.*, 12:447–466, November 2010.

[15] Jean-Raymond Abrial, Michael J. Butler, Stefan Hallerstede, and Laurent Voisin. An Open Extensible Tool Environment for Event-B. In *ICFEM*, pages 588–605, 2006.

[16] Jean-Raymond Abrial and Stefan Hallerstede. Refinement, Decomposition, and Instantiation of Discrete Models: Application to Event-B. *Fundam. Inform.*, 77(1-2):1–28, 2007.

[17] Bowen Alpern and Fred B. Schneider. Recognizing Safety and Liveness. *Distributed Computing*, 2(3):117–126, 1987.

[18] Rajeev Alur. Timed automata. In *CAV*, pages 8–22, 1999.

[19] Rajeev Alur, Costas Courcoubetis, and David Dill. Model-checking in dense real-time. *Information and Computation*, 104:2–34, 1993.

[20] Rajeev Alur, Costas Courcoubetis, and David L. Dill. Model-checking for real-time systems. pages 414–425, 1990.

[21] Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, April 1994.

[22] R. J.R. Back. Refinement of Parallel and Reactive Programs. Technical report, Pasadena, CA, USA, 1992.

[23] Ralph-Johan Back. On the Correctness of Refinement Steps in Program Development. Ph.D. thesis Report A-1978-4, Department of Computer Science, University of Helsinki, Helsinki, Finland, Oct 1978.

[24] Ralph-Johan Back. Refinement Calculus, Part II: Parallel and Reactive Programs. In *REX Workshop*, pages 67–93, 1989.

[25] Ralph-Johan Back and Reino Kurki-Suonio. Decentralization of Process Nets with Centralized Control. In *PODC*, pages 131–142, 1983.

[26] Ralph-Johan Back, Luigia Petre, and Ivan Porres. Generalizing action systems to hybrid systems. In *FTRTFT*, pages 202–213, 2000.

[27] Ralph-Johan Back, Luigia Petre, and Ivan Porres. Continuous action systems as a model for hybrid systems. *Nord. J. Comput.*, 8(1):2–21, 2001.

[28] Ralph-Johan Back and Kaisa Sere. Stepwise Refinement of Action Systems. *Structured Programming*, 12(1):17–30, 1991.

[29] Ralph-Johan Back and Joakim von Wright. Refinement Calculus, Part I: Sequential Nondeterministic Programs. In *REX Workshop*, pages 42–66, 1989.

[30] Ralph-Johan Back and Joakim von Wright. Trace Refinement of Action Systems. In *CONCUR*, pages 367–384, 1994.

[31] Ralph-Johan Back and Joakim von Wright. *Refinement calculus - a systematic introduction*. Undergraduate texts in computer science. Springer, 1999.

[32] Ralph-Johan Back and Qiwen Xu. Refinement of fair action systems. *Acta Inf.*, 35(2):131–165, 1998.

[33] R.J.R. Back. *Correctness preserving program refinements: proof theory and applications*. Mathematical Centre tracts. Mathematisch Centrum, 1980.

[34] Gerd Behrmann, Alexandre David, Kim G. Larsen, Oliver Mller, Paul Pettersson, and Wang Yi. UPPAAL - present and future. In *Proc. of 40*th *IEEE Conference on Decision and Control*. IEEE Computer Society Press, 2001.

[35] J. Berk. *Systems Failure Analysis*. Asm International, 2009.

[36] Robert V. Binder. *Testing object-oriented systems: models, patterns, and tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[37] Jeremy W. Bryans, John S. Fitzgerald, Alexander Romanovsky, and Andreas Roth. Patterns for Modelling Time and Consistency in Business Information Systems. In *ICECCS*, pages 105–114, 2010.

[38] Rod M. Burstall. Program Proving as Hand Simulation with a Little Induction. In *IFIP Congress'74*, pages 308–312, 1974.

[39] Rod M. Burstall and John Darlington. A Transformation System for Developing Recursive Programs. *Journal of the ACM*, 24(1):44–67, 1977.

[40] Michael Butler. Decomposition Structures for Event-B. In *Integrated Formal Methods iFM2009, Springer, LNCS 5423*, volume LNCS. Springer, February 2009.

[41] Michael Butler and Jerome Falampin. An Approach to Modelling and Refining Timing Properties in B. In *Refinement of Critical Systems (RCS)*, January 2002.

[42] Michael Butler and Divakar Yadav. An incremental development of the mondex system in event-b. *Formal Aspects of Computing*, 20(1):61–77, January 2008.

[43] M.J. Butler. A CSP Approach to Action Systems. 1992.

[44] Dominique Cansell, Dominique Méry, and Joris Rehm. Time Constraint Patterns for Event-B Development. In Olga Kouchnarenko Jacques Julliand, editor, *B 2007: Formal Specification and Development in B 7th International Conference of B Users, January 17-19, 2007*, volume 4355 of *Lecture Notes in Computer Science*, pages 140–154, Besançon France, 2007. Springer-Verlag. ISSN : 0302-9743 (Print) ; 1611-3349 (Online) ; ISBN : 978-3-540-68760-3.

[45] Karlis Cerans, Jens Chr. Godskesen, and Kim Guldstrand Larsen. Timed modal specification - theory and tools. In *CAV*, pages 253–267, 1993.

[46] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, 8(2):244–263, April 1986.

[47] Edmund Clarke. Model checking. In S. Ramesh and G Sivakumar, editors, *Foundations of Software Technology and Theoretical Computer Science*, volume 1346 of *Lecture Notes in Computer Science*, pages 54–56. Springer Berlin / Heidelberg, 1997. 10.1007/BFb0058022.

[48] Edmund Clarke and E. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In Dexter Kozen, editor, *Logics of Programs*, volume 131 of *Lecture Notes in Computer Science*, pages 52–71. Springer Berlin / Heidelberg, 1982. 10.1007/BFb0025774.

[49] A. David, M.O. Möller, and Dansk Grundforskningsfond. BRICS. *From HUPPAAL to UPPAAL: a translation from hierarchical timed automata to flat timed automata*. BRICS report series. BRICS, 2001.

[50] J. Davies and Oxford University Computing Laboratory. Programming Research Group. *Specification and proof in real-time systems*. Oxford Technical Monograph PRG. Oxford University Computing Laboratory, Programming Research Group, 1991.

[51] Jim Davies and Steve Schneider. A Brief History of Timed CSP. *Theor. Comput. Sci.*, 138(2):243–271, 1995.

[52] W.P. De Roever and K. Engelhardt. *Data Refinement: Model-Oriented Proof Methods and Their Comparison*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 2008.

[53] Edsger W. Dijkstra. Notes on structured programming. In *Structured Programming*, chapter 1, pages 1–82. Academic Press, London, 1972.

[54] Edsger W. Dijkstra. *A Discipline of Programming*. Prentice Hall, Inc., October 1976.

[55] Edsger W. Dijkstra. *Executional abstraction*. Prentice-Hall, 1976.

[56] Jin Song Dong, Ping Hao, Shengchao Qin, Jun Sun, and Wang Yi. Timed automata patterns. *IEEE Trans. Softw. Eng.*, 34(6):844–859, 2008.

[57] Roger Duke, Gordon Rose, and Graeme Smith. Object-z: a specification language advocated for the description of standards. *COMPUTER STANDARDS AND INTERFACES*, 17, 1995.

[58] E. Durr and J. van Katwijk. VDM++, a formal specification language for object-oriented designs. In *CompEuro '92 . 'Computer Systems and Software Engineering',Proceedings.*, pages 214 –219, may 1992.

[59] Edsger W. Dijkstra. Guarded Commands, Nondeterminacy and Formal Derivation of Programs. *Commun. ACM*, 18(8):453–457, 1975.

[60] Neil Evans and Steve Schneider. Analysing Time Dependent Security Properties in CSP Using PVS. In *ESORICS*, pages 222–237, 2000.

[61] Asieh Salehi Fathabadi. *An Approach to Atomicity Decomposition in the Event-B Formal Method*. PhD thesis, Electronic and Software Systems Group, June 2012.

[62] Asieh Salehi Fathabadi and Michael Butler. Applying Event-B Atomicity Decomposition to a Multi Media Protocol. In *FMCO*, pages 89–104, 2009.

[63] W.H.J. Feijen, A.J.M. van Gasteren, D. Gries, and J. Misra. *Beauty Is Our Business: A Birthday Salute to Edsger W. Dijkstra*. Monographs in Computer Science. Springer, 1990.

[64] Erich Gamma and Kent Beck. *Contributing to Eclipse: Principles, Patterns, and Plugins*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 2003.

[65] Marie-Claude Gaudel. Formal methods and testing: Hypotheses, and correctness approximations. In *FM*, pages 2–8, 2005.

[66] Susan L. Gerhart. Correctness-preserving program transformations. In *POPL*, pages 54–66, 1975.

[67] Stefan Hallerstede. On the Purpose of Event-B Proof Obligations. In *Proceedings of the 1st international conference on Abstract State Machines, B and Z*, ABZ '08, pages 125–138, Berlin, Heidelberg, 2008. Springer-Verlag.

[68] Thomas A. Henzinger, Xavier Nicollin, Joseph Sifakis, and Sergio Yovine. Symbolic model checking for real-time systems. *Information and Computation*, 111:394–406, 1992.

[69] Anders Hessel, Kim Guldstrand Larsen, Marius Mikucionis, Brian Nielsen, Paul Pettersson, and Arne Skou. Testing Real-Time Systems Using UPPAAL. pages 77–117, 2008.

[70] Thai Son Hoang, Alexei Iliasov, Renato Silva, and Wei Wei. A Survey on Event-B Decomposition. *ECEASST*, 46, 2011.

[71] C. A. R. Hoare. An axiomatic basis for computer programming. *COMMUNICATIONS OF THE ACM*, 12(10):576–580, 1969.

[72] C. A. R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1(4):271–281, December 1972.

[73] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[74] Gerard J. Holzmann. Basic SPIN Manual. Technical report, 1994.

[75] Gerard J. Holzmann, Gerard J. Holzmann, and Gerard J. Holzmann. Tutorial: Design and validation of protocols. *Tutorial Computer Networks and ISDN Systems*, 25:981–1017, 1991.

[76] Michael Huth and Mark Dermot Ryan. *Logic in computer science - modelling and reasoning about systems (2. ed.)*. Cambridge University Press, 2004.

[77] D.M. Jackson, University of Oxford. Board of the Faculty of Mathematical Sciences, University of Oxford. Mathematical, and Physical Sciences Division. *Logical verification of reactive software systems*. Oxford University, 1992.

[78] M.A. Jackson. *Michael Jackson System Development*. Englewood Cliffs, N.J. : Prentice/Hall., New York, NY, USA, 1983.

[79] Pankaj Jalote. *An integrated approach to software engineering*. Springer-Verlag New York, Inc., New York, NY, USA, 1991.

[80] Cliff B. Jones. *Systematic Software Development using VDM*. Prentice-Hall, Upper Saddle River, NJ 07458, USA, 1990.

[81] H. Kopetz. *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Real-Time Systems. Springer, 2011.

[82] Leslie Lamport. A Simple Approach to Specifying Concurrent Systems. *Commun. ACM*, 32(1):32–45, 1989.

[83] Leslie Lamport. The temporal logic of actions. *ACM Trans. Program. Lang. Syst.*, 16(3):872–923, 1994.

[84] Kim G. Larsen, Marius Mikucionis, and Brian Nielsen. Online Testing of Real-Time Systems Using UPPAAL: Status and Future Work. In Ed Brinksma, Wolfgang Grieskamp, and Jan Tretmans, editors, *Perspectives of Model-Based Testing*, number 04371 in Dagstuhl Seminar Proceedings, Dagstuhl, Germany, 2005. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany.

[85] Kim Guldstrand Larsen, Paul Pettersson, and Wang Yi. UPPAAL in a Nutshell. *STTT*, 1(1-2):134–152, 1997.

[86] Peter Gorm Larsen, John S. Fitzgerald, and Sune Wolff. Methods for the Development of Distributed Real-Time Embedded Systems Using VDM. *Int. J. Software and Informatics*, 3(2-3):305–341, 2009.

[87] Magnus Lindahl, Paul Pettersson, and Wang Yi. Formal design and analysis of a gear controller. *STTT*, 3(3):353–368, 2001.

[88] Brendan Mahony and Jin Song Dong. Blending Object-Z and Timed CSP: an introduction to TCOZ. In *Proceedings of the 20th international conference on Software engineering*, ICSE '98, pages 95–104, Washington, DC, USA, 1998. IEEE Computer Society.

[89] Zohar Manna and Amir Pnueli. Adequate Proof Principles for Invariance and Liveness Properties of Concurrent Programs. *Sci. Comput. Program.*, 4(3):257–289, 1984.

[90] Zohar Manna and Amir Pnueli. *The temporal logic of reactive and concurrent systems*. Springer-Verlag New York, Inc., New York, NY, USA, 1992.

[91] Carroll Morgan. Data refinement by miracles. *Inf. Process. Lett.*, 26(5):243–246, 1988.

[92] Carroll Morgan. *Programming from specifications*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1990.

[93] Ben C. Moszkowski. *Executing temporal logic programs*. Cambridge University Press, 1986.

[94] Marcel Oliveira, Ana Cavalcanti, and Jim Woodcock. A utp semantics for *circus*. *Formal Asp. Comput.*, 21(1-2):3–32, 2009.

[95] Joël Ouaknine, Alexander Rabinovich, and James Worrell. Time-bounded verification. In *CONCUR 2009: Proceedings of the 20th International Conference on Concurrency Theory*, pages 496–510, Berlin, Heidelberg, 2009. Springer-Verlag.

[96] Joël Ouaknine and Steve Schneider. Timed CSP: A Retrospective. *Electronic Notes in Theoretical Computer Science*, 162:273–276, September 2006.

[97] Joël Ouaknine and James Worrell. Timed CSP = Closed Timed Safety Automata. *Electr. Notes Theor. Comput. Sci.*, 68(2):142–159, 2002.

[98] Jan Peleska. Design and verification of fault tolerant systems with CSP. *Distrib. Comput.*, 5(2):95–106, 1991.

[99] Amir Pnueli. The Temporal Semantics of Concurrent Programs. In *Semantics of Concurrent Computation*, pages 1–20, 1979.

[100] Amir Pnueli. Specification and Development of Reactive Systems (Invited Paper). In *IFIP Congress*, pages 845–858, 1986.

[101] G. M. Reed and A. W. Roscoe L. A timed model for communicating sequential processes. In *Theoretical Computer Science*, pages 314–323, 1988.

[102] Theo C. Ruys. SPIN Tutorial: How to Become a SPIN Doctor. In *SPIN*, pages 6–13, 2002.

[103] J.W. Sanders. *An introduction to CSP*. Oxford Technical Monograph PRG. Oxford University Computing Laboratory, Programming Research Group, 1988.

[104] Mohammad Reza Sarshogh and Michael Butler. Specification and refinement of discrete timing properties in event-b. *ECEASST*, 46, 2011.

[105] S.A. Schneider, University of Oxford. Mathematical, Physical Sciences Division, and University of Oxford. Board of the Faculty of Mathematical Sciences. *Correctness and communication in real-time systems*. University of Oxford, 1989.

[106] Steve Schneider. *The B-Method: An Introduction*. Cornerstones of Computing. Palgrave Macmillan, October 2001.

[107] Renato Silva, Carine Pascal, T. Son Hoang, and Michael Butler. Decomposition Tool for Event-B. In *Workshop on Tool Building in Formal Methods - ABZ Conference*, January 2010.

[108] R.A. Sørensen and J.M. Nygaard. *Evaluating Distributed Architectures Using VDM++Real-Time Modeling with a Proof of Concept Implementation*. Aarhus Universitet, 2007.

[109] P. Tabuada and G.J. Pappas. Linear time logic control of discrete-time linear systems. *Automatic Control, IEEE Transactions on*, 51(12):1862 –1877, dec. 2006.

[110] Stavros Tripakis and Costas Courcoubetis. Extending Promela and Spin for Real Time. In *TACAS*, pages 329–348, 1996.

[111] Marcel Verhoef. *Modeling and Validating Distributed Embedded Real-Time Control Systems*. PhD thesis, Radboud University Nijmegen, 2008. ISBN 978-90-9023705-3.

[112] Niklaus Wirth. Program development by stepwise refinement. *Communications of the ACM*, 14(4):221–227, April 1971.

[113] J. Woodcock and J. Davies. *Using Z—Specification, Refinement, and Proof*. Series in Computer Science. Prentice Hall International, 1996.

[114] Jim Woodcock and Ana Cavalcanti. The semantics of circus. In *ZB*, pages 184–203, 2002.

[115] Sanaz Yeganefard, Michael Butler, and Abdolbaghi Rezazadeh. Evaluation of a guideline by formal modelling of cruise control system in Event-B. In César Muñoz, editor, *Proceedings of the Second NASA Formal Methods Symposium (NFM 2010), NASA/CP-2010-216215*, pages 182–191, Langley Research Center, Hampton VA 23681-2199, USA, April 2010. NASA.

[116] Wang Yi. Real-Time Behaviour of Asynchronous Agents. In *CONCUR*, pages 502–520, 1990.

[117] Sergio Yovine. Kronos: A verification tool for real-time systems. *STTT*, 1(1-2):123–133, 1997.