Modelling Complex Timing Requirements with Refinement

Gintautas Sulskus, Michael Poppleton
and Abdolbaghi Rezazadeh
University of Southampton, UK
gs6g10 | mrp | ra3@soton.ac.uk

Abstract

In the domain of formal modelling and verification of real-time safety-critical systems, our focus is on complex – i.e. nested, interdependent and cyclic – timing constraints. In Event-B, we present methodological support for our concept of timing interval by defining a set of refinement transformations, designed for structured modelling of such timing constraints. All timing interval related aspects are generated by our tool. An example development, abstracted from our work modelling a cardiac pacemaker, serves to illustrate the use of the transformations. The development is undertaken, proved and model-checked in the Rodin tool-kit for Event-B.

1 Introduction

A number of factors contribute to the complexity and challenge of real-time control systems. Concurrent and communicating components can exhibit unpredictable interactions that may lead to incorrect behaviours. Formal methods are used for rigorous modelling and verification of safety-critical real-time systems. Mathematical models help to eliminate ambiguities in the requirements and enable generation of verification conditions which then can be proved using theorem provers.

This work is a result of our work on a cardiac pacemaker case study, based on [1]. The pacemaker is a complex control system that interacts with the non-deterministic heart via sensors and actuators. The pacemaker’s functionality depends on its internal model of a normal heart. The normal heart is modelled in terms of a set of interdependent nested cyclic timing intervals, representing various requirements of the normal pacing cycle [7]. The model describes electrical activity in each of two pacemaker channels, representing the atrial and ventricular heart chambers respectively. A channel may sense an intrinsic electrical signal from the heart, resulting in contraction of its chambers. A channel may be subjected to a pace actuating signal from the pacemaker to initiate contraction, in the absence of a timely sensed signal. These and similar events define the bounds of the modelled timing intervals. Our example will consider just one of these intervals, the Atrio-Ventricular Interval (AVI), triggered by an atrial event and responded (terminated) by a ventricular event.

This work develops the timing interval concept in a tooled refinement framework absent in other timing approaches. The concept allows to formally relate and reason about multiple requirements. We use the Rodin tool [3] to formally model and elaborate timing intervals in the Event-B formalism [5] through a set of refinements. Timing intervals and their refinements are specified with tiGen – our timing interval modelling tool. Finally, we use tiGen to automatically generate the corresponding Event-B code from the specification.

Our contribution is a set of refinement transformations that build on our timing interval approach [16], and the tiGen tool that facilitates the use thereof. The transformations enable further elaboration of the timing interval in a methodical and reusable manner. The transformations can be combined together and are independent of the application logic of the model. tiGen provides an editor to specify and validate the timing interval and its refinement relation. The tool generates a corresponding Event-B code from the given specification. Here we focus on generative refinement transformations for timing interval refinement.

We present a number of improvements over our previous work. Since the pacemaker is an infinitely cyclic system, we take two steps to improve verification with finite-state model-checkers. Firstly, we address the infinite state-space issue by switching from absolute to relative timing. Secondly, the original work allows unbounded proliferation of housekeeping data for intervals. We address this by the reuse of such data.

Section 2 introduces Event-B and the notion of timing interval. In Section 3 we explain the underlying timing interval semantics in a small example and present the refinement transformations of the interval approach by refining the given example model. Section 4 presents the verification and validation results. In Section 5 we contrast ours with related work. Section 6 concludes.
2 Preliminaries

The Event-B [5] formalism has evolved from the Classical B-Method [4]. Most of the formal concepts it uses, such as guarded actions and refinement were already proposed in Action Systems [3]. Event-B focuses on reactive systems and is aimed at closed system modelling whereas Classical B focuses on software. We prefer Event-B for its simplicity of notations, extensibility and tool support.

An Event-B model is composed of contexts and machines. Contexts specify the static part of a model such as carrier sets, constants and axioms. Machines represent the dynamic part of a model and contain variables, invariants and events. Each machine may refer to one or more contexts. The event is the mechanism that changes state. An event takes parameters, and only when all its guard predicates are true, uses its actions to update state variables. Conceptually, events in Event-B are atomic and instantaneous.

Event-B employs proof-based verification. The system’s safety requirements are specified using invariant properties from which Event-B verification conditions, called proof obligations (POs), are then generated.

Refinements in Event-B takes two forms. Superposition refinement introduces new requirements and structure. Data refinement brings the model closer to implementation by elaborating data structures and algorithms.1 Refining variables are linked to abstract variables by means of gluing invariants whose associated POs ensure correctness of the refinement.

The main platform for Event-B development is the Rodin [3] tool. It is an Eclipse based IDE that provides effective support for modelling, refinement and proof. Rodin auto-generates POs. These are then discharged by automated theorem provers, such as AtelierB [2] or SMT [11], or manually via the interactive proving environment. Rodin provides a wide range of plug-ins, such as the Camille text editor and the ProB [12] finite-state animator and model-checker.

2.1 Timing Interval

We build our contribution of timing interval refinement transformations on our timing interval approach [16]. The timing interval [1] is a higher level abstraction that builds on Sarshogh’s original Delay and Deadline timing properties [14]. Our approach is designed for managing intervals that may be interconnected, nested and cyclic. The timing interval is characterised by one or two timing properties and a set of events – optional ones denoted by [ ], The system may have a number of timing intervals, each identified by a unique name. There may be multiple active instances of a given interval that occur independently from each other [16].

\[ \text{Int}_{\text{name}}(T_1)[...T_i]; R_1[...R_j]; [I_1...I_k]; TP_1(t_1), TP_2(t_2)) \] (1)

The interval is composed of three kinds of events. One of a set of trigger events \( T \in T_1..T_i \) always creates a new instance of the named interval. One of a set of response events \( R \in R_1...R_j \) always terminates an interval instance under conditions specified by timing properties. If there is no active interval instance to terminate, the response event is disabled. In order to be well defined, the interval must have at least one trigger and one response event. One of a set of optional abort events \( A \in A_1..A_k \) aborts the interval. Unlike the response event, the abort event is not constrained by timing properties \( TP \) and does not block if there is no active interval instance to abort. Such behaviour allows the abort event to perform state updates apart from managing the timing interval. The abort event always aborts an active interval instance (if one exists).

The interval must have at least one timing property \( TP(t) \) of duration \( t \), where \( TP \) stands for Deadline or Delay. Deadline means that a response event must occur within a specified time \( t \) of a trigger event occurring. In case of Delay, the response event cannot occur before time \( t \) of trigger event occurring. The interval can have one of three TP configurations: (i) Deadline; (ii) Delay; (iii) Delay and Deadline. If both timing properties are specified, the delay duration must be less or equal to the deadline duration \( (I_{\text{Delay}} \leq I_{\text{Deadline}}) \). We permit equal delay and deadline durations in order to be able to designate a specific point in time.

3 Timing Interval Refinement

We start this section by using the abstract machine of the example model [19] to explain the underlying Event-B semantics and the dynamics of the timing interval [16][2]. Interval approach improvements – index recycling and relative timing – are briefly covered on the way. We then present the refinement transformations in an example model of the AVI mechanism. The refinement structure of the provided example model corresponds to that of our pacemaker case study, unless stated otherwise. The refinement transformations are explained through three model refinements.

The AVI is the interval between the sensed or paced atrial event and the next sensed or paced ventricular event. When active, AVI is constrained by delay AVI\(_{DDL}\) and deadline AVI\(_{DDL}\) timing properties. The timing property durations are specified as constants AVI\(_{DDL} \leq AVI_{DDL} \) respectively.

The interval can be aborted by a sensed ventricular event at any time after it triggers, otherwise the pacemaker delivers a ventricular pace after a time between AVI\(_{DDL} \) and

---

1When discussing a refinement \( B \) of a model \( A \), we may refer to \( A \) as the “abstract” and \( B \) as the “concrete” model.

2In this section we give symbolic names to the defining formulas of the timing interval for reference in later sections.
AVI_JDDL. In certain circumstances (sections 3.2, 3.3) a ventricular pace may occur before AVI_JDLY.

The interval (Figure 1) is triggered by either sensed or paced atrial activity represented by event ax and is responded to by delivering a ventricular pace represented by event vp. Abort event vx – representing either sensed or paced ventricular activity – is always enabled.

The work-flow to model a timing interval comprises two steps – specification and code generation. We use the tiGen tool to specify timing interval in the format of (1); T, R and A roles are assigned to existing model events; TPs and their durations are specified. The tool then automatically generates Event-B code for the timing intervals from the given tiGen specification. The Event-B code for interval AVI is generated from the following specification: AVI(ax; vp; vx; AVI_JDLY, AVI_JDDL).

The tiGen tool code generation is based on generic code templates and comprises several steps [10]: (i) each specification element is assigned a predefined generic Event-B code template, (ii) the generic code variables are instantiated by prefixing them with the interval name, and (iii) the instantiated code is then injected into the model locations based on the specification and template-specific injection rules.

An interval may have in general multiple instances, thus a set of identifying instance indices is used. Because the pacemaker uses single instance intervals, we simplify matters here by using only one instance called IDX.

Clock variable AVI_clocks records the time elapsed from the start of an interval instance, as long as it is active. AVI_clocks is updated in the tick event that models the flow of time. The update action states that for each interval instance, that has not been responded to or aborted, the clock is incremented and the old values are overwritten.

The work-flow to model a timing interval comprises two steps – specification and code generation. We use the tiGen tool to specify timing interval in the format of (1); T, R and A roles are assigned to existing model events; TPs and their durations are specified. The tool then automatically generates Event-B code for the timing intervals from the given tiGen specification. The Event-B code for interval AVI is generated from the following specification: AVI(ax; vp; vx; AVI_JDLY, AVI_JDDL).

Figure 1 introduces, from left to right, the variables associated with trigger, abort and response events and the clock variable that stores interval’s progress. When interval AVI is triggered by executing event ax for the first time, a new index IDX is added to the trigger index set variable AVI_trig. Maplet IDX → 0 is recorded in a clock variable AVI_clocks. After the interval is responded to or aborted, the index is added to either index set variable AVI_resp or AVI_abrt respectively. Correspondingly, function variables AVI_trig_ts and AVI_resp_ts record interval instance trigger and response occurrence timestamps. On the firing of a subsequent trigger, the index is reused - added to AVI_trig and deleted from each of AVI_resp and AVI_abrt. This behaviour is subject to invariants

\[
AVI_{\text{resp}} \cap AVI_{\text{abrt}} = \emptyset \\
AVI_{\text{resp}} \cup AVI_{\text{abrt}} \subseteq AVI_{\text{trig}} \quad (IDX_{\text{INV}})
\]

\[
\text{forall } \text{id} \in AVI_{\text{trig}} \land id \notin AVI_{\text{resp}} \Rightarrow AVI_{\text{clocks}}(id) \leq AVI_{\text{trig}}_{\text{ts}}(id) + AVI_{\text{JDLY}} \quad (DLY_{\text{INV}})
\]

\[
\text{forall } \text{id} \in AVI_{\text{trig}} \land id \in AVI_{\text{resp}} \Rightarrow AVI_{\text{clocks}}(id) \geq AVI_{\text{trig}}_{\text{ts}}(id) + AVI_{\text{JDDL}} \quad (DLY_{\text{GRD}})
\]

The deadline timing property consists of two invariants. Invariant \( (DLY_{\text{INV}}) \) expresses the requirement that while the interval instance is active, it must not exceed the deadline duration AVI_JDDL. \( (DLY_{\text{INV}}) \) specifies that the active interval AVI instance must be responded to within duration AVI_JDDL of the trigger event occurring.

\[
\text{forall } \text{id} \in AVI_{\text{trig}} \land id \notin AVI_{\text{resp}} \land \text{id} \notin AVI_{\text{abrt}} \Rightarrow AVI_{\text{clocks}}(id) + 1 \leq AVI_{\text{trig}}_{\text{ts}}(id) + AVI_{\text{JDDL}} \quad (DLY_{\text{GRD}})
\]

This defines the formal structure of a delay-deadline interval, which is generic modulo its refinement transformation definition. All refined intervals have this structure except where otherwise indicated.

Notice, that variable AVI_clocks cannot progress beyond AVI_JDDL due to the deadline timing property constraint. This and the index reuse upon triggering allows us to create cyclic models with finite variables and sets that can be then fully model-checked by ProB.

### 3.1 Alternative Interval Transformation

For each refinement transformation, we describe its purpose and structure as generic code templates.
The first refinement adds a finer grained view of the AVI interval dynamics. In particular, we distinguish interval AVI as either initiated by the actuated atrial pace event or as the sensed intrinsic atrial event – intervals pAVI and sAVI respectively. The dynamics of the intervals are different, and therefore are their parameters. In this example, refined intervals pAVI and sAVI are treated differently in terms of their duration constraints [7]. The alternative refinement transformation refines an abstract interval to two or more alternative intervals.

Only one of the alternative interval instances can be active at a time; in this example, either pAVI or sAVI. Three additional requirements apply: (i) alternative intervals pAVI and sAVI cannot share the same trigger or response event; (ii) the alternative intervals must have the same timing properties as the abstract timing interval, and (iii) the property durations must be consistent. The consistency of the timing property durations between intervals AVI and pAVI is ensured with two axioms [2] and [3], and similarly for sAVI. We mark new constants and variables in bold in all further equations.

\[ p_{AVI_{DLY}} \geq AVI_{DLY} \quad (2) \quad p_{AVI_{DOL}} \leq AVI_{DOL}. \quad (3) \]

Interval pAVI (Figure 2) is triggered by the paced atrial stimulus, represented by event ap, whereas interval sAVI is triggered by the sensed intrinsic atrial activity, represented by event as. Both events ap and as refine the abstract event ax. Intervals pAVI and sAVI can be either aborted by vx or responded to by p_vp and s_vp respectively. Both response events refine event vp.

At the Event-B level, we distinguish alternative interval pAVI and sAVI instances by refining the abstract trigger variable AVI_trig to new variables pAVI_trig and sAVI_trig, subject to (4). New constants are marked in bold.

\[ \text{partition}(AVI_{trig}, p_{AVI_{trig}}, s_{AVI_{trig}}) \quad (4) \]

The partitioning ensures that the indices are mutually exclusive with respect to alternative intervals pAVI and sAVI. Similarly, the abstract response variable AVI_resp is partitioned to variables pAVI_resp and sAVI_resp. All new invariants, guards and actions, related to interval pAVI, rely on the concrete variables pAVI_trig and pAVI_resp instead of the abstract ones (Figure 2). These new variables are marked in bold; reused ones (e.g. AVI_abrt) are in plain text. We will use this convention in all such figures. The dotted line marks the separation between the intervals.

![Figure 2: Interval pAVI](image)

The given code examples for alternative interval pAVI, and further refinements as well, have been automatically instantiated from the generic code templates by the tiGen tool. The principle of code generation is analogous to that used to generate timing interval AVI (Figure 1). Duration consistency axioms, such as (2) and (3), are generated similarly to other templates, and thus are not covered.

The invariant template (Figure 3) defines trigger and response variables for n alternative intervals resulting in code such as (4). The template is applied once as specified by the rule #gen. Prefix P_ is a place holder for the concrete alternative interval name that gets instantiated for x \(\in 1..n\). P_trig and P_resp are the abstract interval’s trigger and response variables.

We define Event-B code templates for alternative interval trigger and response event types (Figure 4). The templates consist of parameters, guards and actions that are needed for a specific interval role. Event templates are instantiated once for each alternative interval. The instantiated code is then injected to all #target interval events.

Event T_1 represents a trigger event template for x_trig, where the latter are trigger events of a specific alternative interval. This template adds a new index to the interval’s trigger variable P_s_trig and clears it from the response variable P_s_resp. Place holder p_P_trig is instantiated with the abstract interval’s trigger parameter. In this example, T_1 is instantiated and injected twice – once for the trigger event ap of pAVI and once for the trigger event as of sAVI. The trigger template T_2 is injected to all events r_trig \(\setminus x_{trig}\), where r_trig represents concrete trigger events of the most abstract (root) timing interval refinement of x. In this case the root timing interval is AVI and the concrete trigger events for it are ap and as (both refine ax). In case of pAVI, the template is injected to event as, and in case of sAVI, it is injected to ap. The injected code ensures that the interval index can be reused by any of the alternative intervals. p_P_trig is defined by abstract interval’s guards and carries indexes that are new, aborted or responded to.

Event template R injects a guard and an action to response events x_resp of a specific interval. The guard ensures that P_s_trig contains the index of the timing interval instance to be responded to. The action records the response. p_P_resp is replaced with the response parameter of the abstract event.

The template of the deadline timing property is instantiated for each concrete timing interval. The template consists of two invariants and a guard in Tick event (Figure 6). Invariants i1 and i2 express the requirement of the deadline timing property analogous to (DDL_INV_1) and (DDL_INV_2) respectively. Place holders P_resp_ts, P_trig Ts and P_clocks correspond to trigger and response timestamps and the clock variable, all of which relate to the abstract event. Guard g1 analogous to (DDL_GRD), is for Tick
event. Rule #1 in Tick event instructs to remove any existing deadline guard of the abstract interval. This generates the PO that the newly generated deadline guards for the alternative intervals \(1 \leq n \leq m\) preserve refinement correctness.

The template for the delay timing property (Figure 5) consists of a single invariant \(i1\) and a guard \(g1\) on the response event. The template is analogous to invariant \((\text{DLY}_\text{INV})\) and guard \((\text{DLY}_\text{GRD})\). Like the deadline template, it is generated once for each alternative interval.

### 3.2 Subinterval Transformation

After the start of \(pAVI\), there is a minimum period, the Ventricular Safety Period (VSP), strictly shorter than the AVI, during which a paced ventricular event is prohibited [7]. We introduce the first of two refinement transformations to implement this. By refinement we divide interval \(pAVI\) into a sequence of two subintervals VSP and VSPo (“\(p\)” stands for “off”).

The subinterval refinement transformation refines an abstract interval to a sequence of two subintervals. The order of subinterval occurrence is strict. Requirements are that (i) the first subinterval must have the same trigger events as the abstract interval, (ii) the last subinterval must have the same response events as the abstract interval, (iii) a preceding subinterval’s response event must serve as trigger event for the succeeding subinterval, and (iv) the sum of all subinterval delay durations and the sum of all subinterval deadline timing property durations must be consistent with those of the abstract interval. For (iv), two axioms are generated: [5] ensures that the sum of subinterval delay durations is greater or equal to that of the abstract interval, and [6] is similar for deadline timing property durations.

\[
\begin{align*}
\text{VSP}_\text{JOLY} + \text{VSPo}_\text{JOLY} &\geq \text{AVI}_\text{JOLY} \quad (5) \\
\text{VSP}_\text{JDDL} + \text{VSPo}_\text{JDDL} &\leq \text{AVI}_\text{JDDL} \quad (6)
\end{align*}
\]

Subinterval VSP (Figure 7) is triggered by event \(ap\). New event \(\text{vsp\_ended}\) serves simultaneously as the response event for interval VSP and as the trigger event for interval VSPo. Interval VSPo is responded to by event \(p\_vp\). Both intervals may be aborted by event \(vx\).

Figure 7 lists variables for subintervals VSP and VSPo. Interval VSP reuses abstract trigger variables. New intermediate variables \(\text{VSP\_resp\_VSPo\_trig}\) \(\leq \text{pAVI}\_trig\) and corresponding timestamp \(\text{VSP\_resp\_VSPo\_trig\_ts}\) record the occurrence of both VSP response and VSPo trigger events.

The abstract abort variable is reused by both intervals.

The process of adding delay and deadline timing property invariants for subinterval VSP is similar to that of \(pAVI\). The abstract clock variable is used to record the total duration of both subintervals. When interval VSP is responded to, the clock is not reset, but continues to progress for the duration of the subinterval sequence. The timestamp, recorded when event \(\text{vsp\_ended}\) fires, is then used as an offset to adapt the delay invariant \((\text{DLY}_\text{INV})\) for VSPo:

\[
\forall \text{idx}\_\text{idx} \in \text{VSP\_resp\_VSPo\_trig} \land \text{idx} \in \text{AVI\_resp} \Rightarrow \text{AVI\_resp\_ts(\text{idx})} \geq \text{VSP\_resp\_VSPo\_trig\_ts(\text{idx})} + \text{VSPo\_JDDL}
\]

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

All sub-interval templates are instantiated once for each interval. We define two template variables \(\text{Strig}\) and \(\text{Sresp}\) (Figure 8) whose value depends on the sub-interval’s position in the sequence. Rule #1 says that if it is the first sub-interval, then \(\text{Strig}\) refers to the abstract interval trigger variable. Otherwise it refers to the trigger-response variable, shared with the preceding sub-interval. Rule #2 says that in case it is the last sub-interval \(n, \text{Sresp}\) refers to the abstract interval response variable, else it refers to the trigger-
response variable, shared with the succeeding sub-interval.

The sub-interval base template (Figure 9) declares the relation between trigger and response variables of a specific sub-interval. Deadline (Figure 11) and delay (Figure 12) code template structure is analogous to that of the alternative interval templates.

We define three templates (Figure 10) for trigger (T), shared trigger-response (R1) and response (R2) events. The instantiated code is injected to target events if a #condition (Figure 10) is satisfied. If not specified, the #condition defaults to true.

Trigger event template T injects Event-B actions that clear shared variables. The template is injected to all events r_trig. The latter is all concrete trigger events of the root timing interval of x. In this case the root interval is AVI and its concrete trigger events are ap and as. The instantiated code is injected only if it is not the last sub-interval in the sequence – x ∈ 1..(n – 1). The last sub-interval, VSPo in this case, uses abstract interval response variables instead of the shared variables. Hence it is excluded from this template.

The shared trigger-response event template R1 adds a shared trigger-response parameter p1. Guards g1 and g2 ensure the parameter carries an index of the active preceding interval. Actions a1 and a2 update the shared variable. The template is applicable for all except the last sub-interval and is injected to all current sub-interval’s response events. In this example the template is injected to the shared event vspEnded and operates on the shared variable vspRespVSPo_trig and its timestamp.

The R2 event template is instantiated and injected to all response events of the last sub-interval. The guard, as shown in R2 template, ensures that the response parameter carries an active but not yet responded to index of the sub-interval n. Finally, we inject the instantiated code of this template to the interval VSPo response event p_vp.

3.3 Abort-To-Response Transformation

We use this transformation to complete the modelling of VSP. If any activity is sensed on the ventricular channel while the VSP interval is active, a ventricular pace must be delivered at the end of the interval and not before [7]. This is achieved by refining the abort event υx of interval VSP - which can fire at any time - into the response vsp_pace. This new event represents the ventricular pace at the end of the VSP subinterval. Note that vsp_pace works as the response event for interval VSPa only locally. The event retains its role as the abort event for the subinterval abstract sequence of VSP and VSPo.

This refinement transformation requires an interval with at least one abort event. It then transforms all abort events to response events. If a transformed abort event serves as an abort for other intervals, it retains that role for them. The refinement transformation provides timing interval elaboration without breaking the timing interval structure thus retaining the possibility for further elaborations.

Abort variable AVI_abrt is no longer used for abort purposes by the VSPa interval and any succeeding refinement of it. The interval reuses abstract trigger variables (Figure 13).

We start the transformation by introducing new response variables that treat abort indices of interval VSPa as the response. The new response variable VSPa_resps is defined as the abstract interval VSP response indices, together with the VSP abort indices [8]. The latter are precisely those indices that are triggered and aborted. Note that variable AVI_abrt holds abort indices for all intervals in the refinement chain, and (pAVI_trig ∩ AVI_abrt) filters out only those indices that are associated with interval VSPa. We then introduce a new timestamp [9], associated to the new response variable.
VSP\_resp = VSP\_resp ∪ (AVI\_trig ∩ AVI\_abort) (8)
VSP\_resp\_ts ∈ VSP\_resp → N (9)

Event \textit{vx}, which previously served as an abort event for interval VSP, is refined for intervals VSPo and sAVI, but removed from the VSP\_resp specification (Figure 13). This is done by injecting a guard (10) to prevent the abortion of VSPa and to preserve invariant (8). Event \textit{vx} parameter p\_AVI\_abort holds the index of the active interval instance to be aborted. The guard ensures that the parameter never holds the index of the active interval VSP\_resp. Event \textit{vx} retains the role as the abort event for intervals VSPo and sAVI.

p\_AVI\_abort ∉ (pAVI\_trig \setminus \textit{VSPa\_resp}) (10)

The transformed response event vsp\_pace (Figure 14) is injected with the guard g1. The guard restricts the abstract abort parameter p\_AVI\_abort to the active but not yet responded to indices of VSP\_resp. Due to this guard, event vsp\_pace can only be executed when VSP\_resp is active. Grd_abrt represents guards that have been generated for the abstract abort event \textit{vx} of interval AVI. Upon execution of event vsp\_pace, the index is recorded to the new response variable (a1) and the timestamp is taken (a2). Act_abrt represents other actions, related to AVI abort role.

\[ i1: \forall idx:idx ∈ pAVI\_trig ∧ idx ∈ \textit{VSPa\_resp} \Rightarrow \text{VSPa\_resp\_ts(idx)} ≥ \text{VSPa\_ts} \]
\[ \text{Event vsp\_pace refines } \textit{vx} = \setminus \text{Grd_abrt} \]
\[ g1: p\_AVI\_abort ∈ pAVI\_trig \setminus \textit{VSPa\_resp} \]
\[ g2: AVI\_clocks[p\_AVI\_abort] ≥ \text{VSPa\_ts} \]
\[ \text{Act_abrt} \]
\[ a1: \textit{VSPa\_resp} := \textit{VSPa\_resp} ∪ \{p\_AVI\_abort\} \]
\[ a2: \textit{VSPa\_resp\_ts} := \textit{VSPa\_resp\_ts} (p\_AVI\_abort) \Rightarrow (p\_AVI\_abort → AVI\_clocks(p\_AVI\_abort)) \]

Figure 14: Evt. vsp\_pace: injected abort-to-response code.

As before for pAVI in subsection 3.1, we generate the delay timing property. Invariant i2 is instantiated over new and reused variables as displayed in (Figure 14). To preserve this invariant, guard g2 is injected to VSP\_resp response events vsp\_end and vsp\_pace. The code generation of the deadline timing property follows similar process as described in subsection 3.1.

Analogous to previous refinement transformations, actions to clear the new response variable (11) and its corresponding timestamp (12) are injected to all concrete trigger events – ap and as, which belong to root timing interval AVI. Parameter p\_AVI\_trig, present in both events, carries the index of the interval to be triggered.

VSP\_atr\_resp := VSP\_atr\_resp \setminus \{p\_AVI\_trig\} (11)
VSP\_atr\_resp\_ts := \{p\_AVI\_trig\} \Leftrightarrow VSP\_atr\_resp\_ts (12)

For simplicity, we treat the parameter as an element. Due to Event-B language specifics, in the actual case studies we model the abort event parameter as a set. The simplification does not change the semantics of the given examples.

\[ \text{Axioms (13) and (14) ensure consistency between the VSP and VSP\_resp interval.} \]

\[ \text{VSPa\_ts} ≥ \text{VSP\_ts} \]

The given code examples were fully generated from the abort-to-response code templates. The principle of generation is similar to previously discussed.

Finally, interval VSP\_resp can be further elaborated with the refinement transformations. However, the index set design means that such concrete intervals and their refinements cannot abort.

4 Verification and Validation

The provided example model has three concrete timing intervals: VSP\_resp, VSP\_o and AVI. All of the 189 timing-related POs were discharged automatically. The use of relative timing and index recycling allowed us to fully model-check our model in the ProB – we found the model to be deadlock-free with no invariant violations. Finally, we used ProB to validate the model by manual animation.

A fuller evaluation of our refinement transformation approach is the pacemaker case study. The timing interval Event-B code with constant timing property durations has been fully generated with tiGen. No customisations were needed to adapt the generated code. In total 11 timing interval refinement transformations were applied subsequently on two timing intervals. The pacemaker model resulted in six refinements with the final refinement having 9 timing intervals. Overall, the model has 144 timing related invariants. There are 1088 time-related proof obligations, all of which were discharged using auto-provers. A full coverage model-check was performed using ProB. We used test case scenarios and a simple heart model simulation engine to further validate our pacemaker model. Model-checking and simulation were successful.

5 Related Work

We consider various approaches in the literature on managing timing requirements via refinement.

Berthing et al. [8] propose a design work-flow with alternating data and timing constraint refinement steps. Step-wise refinement of an Event-B model is guided at each step by transformation to an Uppaal model, which is used for annotation of the next Event-B refinement. The goal of the authors is to take advantage of a refinement support in Event-B and a verification of timing requirements in UPPAAL[6]. The automation of the proposed design transformations remain for future work.

Cansell et al. [10] propose modelling time as a variable \textit{time} ∈ N. An event \textit{post\_time} adds a new future \textit{active time}...
to a variable \( at \subseteq \mathbb{N} \). The advance of time is modelled in terms of processing these active time \( at \) elements. The paper recommends (i) to start with an abstraction including scheduling without time and prove more important abstract properties of the system, and (ii) to introduce timing in a subsequent refinement.

Méry and Singh [13] apply the approach of [10] to model the timing constraints of a single electrode pacemaker system. At the abstract level the fundamental timing intervals are introduced, then gradually enriched through four refinements. The enrichment is case-specific and is coupled with the model structure, thus limiting the potential for reuse. The system was verified by proof (11% of POs were discharged manually) and manually validated with the ProB tool.

Sarshogh [14] introduces the notion of delay and deadline on which we build our work, and provides several associated refinement patterns. The concept of our alternative and subinterval refinement transformations is inspired by these patterns. [14] provides a tool to generate timing aspects, but without the refinement support.

Our timing interval approach [16] is based on the abstract notion of interval and is designed for cyclic interdependent timing constraints. We use interval-specific clocks rather than a single clock (as proposed by [10]) for all intervals in order to address the infinite state-space problem. In contrast to [8], our modelling approach is homogeneous in that we rely on Event-B for modelling both functional and timing aspects.

6 Conclusions and Future Work

In the simple example model we have demonstrated a set of timing interval refinement transformations that gradually elaborate the abstract pacemaker timing interval \( AVI \). We then used our tiGen tool to specify and generate the timing interval and subsequent refinement transformations. The timing interval approach and the refinement transformations have been verified and validated in the given project example and also in the full pacemaker [18] and the landing gear [9][17] case studies. We have established that a subsequent application of the refinement transformations is feasible. Message passing [14] case study is under active development in order to validate the applicability and reusability of our refinement transformations. To date the validation of the case studies yielded similar results to those of section 4.

All timing interval refinements have been proved automatically in both the abstracted pacemaker example demonstrated in this paper and the mentioned case studies. Relative timing and index recycling improvements make a full-coverage model-checking of the approach feasible. We are currently extending the timing interval specification to include a dynamic duration.

Future work includes liveness temporal property verification with the ProB model-checker. We plan to optimise the timing interval approach for the single instance case and potentially a lighter proof burden. We plan to investigate more complex dependencies between several distinct timing intervals in terms of deadlock freedom. Further improvements of the tiGen tool are planned in order to improve usability. Finally, we plan to use a co-simulation tool [15] to validate our pacemaker model against a more sophisticated heart simulation.

References


