An Interval-Based Approach to Modelling Time in Event-B

Gintautas Sulskus, Michael Poppleton, Abdolbaghi Rezazadeh

To cite this version:

Gintautas Sulskus, Michael Poppleton, Abdolbaghi Rezazadeh. An Interval-Based Approach to Modelling Time in Event-B. 6th Fundamentals of Software Engineering (FSEN), Apr 2015, Tehran, Iran. pp.292-307, 10.1007/978-3-319-24644-4_20 . hal-01446607

HAL Id: hal-01446607

https://hal.inria.fr/hal-01446607

Submitted on 26 Jan 2017

An Interval-Based Approach to Modelling Time in Event-B

Gintautas Sulskus, Michael Poppleton and Abdolbaghi Rezazadeh

University of Southampton
{gs6g10,mrp,ra3}@ecs.soton.ac.uk

Abstract. Our work was inspired by our modelling and verification of a cardiac pacemaker, which includes concurrent aspects and a set of interdependent and cyclic timing constraints. To model timing constraints in such systems, we present an approach based on the concept of *timing interval*. We provide a template-based timing constraint modelling scheme that could potentially be applicable to a wide range of modelling scenarios. We give a notation and Event-B semantics for the interval. The Event-B coding of the interval is decoupled from the application logic of the model, therefore a generative design of the approach is possible. We demonstrate our interval approach and its refinement through a small example. The example is verified, model-checked and animated (manually validated) with the ProB animator.

1 Introduction

Control systems must interact with all possible events that the environment may present. A number of factors contribute to the complexity and challenge of these systems. Concurrent and communicating components tend to exhibit unpredictable interactions that may lead to incorrect behaviours. Moreover, timing constraints add real complexity to real-time control systems.

Formal methods are used for rigorous modelling and verification of safety-critical real-time systems. Mathematical models enable generation of verification conditions which then can be proved using theorem provers. Formalising complex real-time systems is demanding, thus suitable modelling abstractions are desirable.

This work emerges from our work on a cardiac pacemaker case study [1]. The pacemaker is a complex control system that interacts with a non-deterministic environment (the heart) via sensors and actuators, whose functionality depends on its internal model of a normal heart. The pacemaker identifies certain heart dysfunctions and intervenes when necessary in order to maintain a correct heartbeat rate. The normal behaviour of the heart is usually modelled [8] in terms of a set of interconnected time intervals, representing various requirements of the normal pacing cycle. The pacemaker intervenes when the heart is observed to violate these requirements. The pacemaker can be single- or dual-channel, being able to interact with one or both (atrium and ventricle) heart chambers respectively.

In this paper we present a timing interval approach that builds on the existing notion of delay, deadline and expiry [21]. We introduce the concept of the interval and reusable patterns that are potentially suitable for modelling systems. Their demands range from a single deadline timing constraint to systems with complex timing constraints that are cyclic, concurrent and interdependent.

A timing interval can have lower and upper boundary timing constraints defined in a number of ways. Typically, such timing constraints share many elements, such as trigger and response events. We present a notation for our timing interval approach that helps describe the timing requirements at a high level but hides the underlying implementation complexity from the modeller.

We demonstrate the interval approach through an example model. The example is modelled in the Event-B language [5] with the Rodin [6] tool. Our development process consists of two main stages. In the first stage, we express the system in UML diagrams using the UML-like modelling tool called iUML [3]. In the second stage, we add explicit timing using our interval approach. We leverage the power of abstraction and reuse via templates.

Section 2 introduces Event-B and the related formal approaches to modelling timing. Section 3 gives the Event-B semantics of the timing interval as a pattern-based collection of variables, invariants, event guards and actions. The approach allows the intervals to be specified in a manner that does not interfere with the logic of the model, and in a compositional fashion. This affords the opportunity to a generative description of the approach with a potential for automated support; in section 4 we give Event-B code templates for such potential automation. In section 5 we give an example of the interval refinement. Sections 6, 7 present verification and validation results of the approach and discuss related work on the pacemaker. Section 8 concludes.

2 Preliminaries

The Event-B [5] formalism is an evolution of the Classical B method [4]. Most of the formal concepts it uses were already proposed in Action Systems [7]. Event-B focuses on reactive systems and is aimed at modelling whereas the Classical B is just for 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 s, constants c and axioms $A(s, c)$. Machines represent the dynamic part of a model and contain variables v, invariants $I(s, c, v)$ and *events*. An event may accept a number of parameters x and consists of at least two blocks: guards $G(x, s, c, v)$ that describe the conditions that need to hold for the occurrence of the event, and actions which determine how specific state variables change as a result of the occurrence of the event. Conceptually, events in Event-B are atomic and instantaneous. Contexts can be extended by other contexts and machines can be refined by other machines. Each machine may refer to one or more contexts.

Event-B employs a strong proof-based verification. The system's safety property requirements are encoded as invariants from which Event-B verification conditions, called *proof obligations* (POs), are then generated. There are various kinds of POs concerned with different proof problems. For instance, an Invariant Preservation PO (INV) indicates that the invariant condition is preserved by an event with before-after predicate R:

$$A(s, c) \land I(s, c, v) \land G(x, s, c, v) \land R(x, s, c, v, v') \vdash i(s, c, v') \tag{1}$$

where $i(s, c, v')$ is a modified specific invariant.

Systems are usually too complex to model all at once. Refinements help to deal with the complexity in a stepwise manner, by developing a system incrementally. There are two forms of refinement in Event-B. The feature augmentation refinement (*horizontal refinement*) introduces new features of the system. The data refinement (*vertical refinement*) enriches the structure of a model to bring it closer to an implementation structure. Refined variables are linked to the abstract layer state variables by means of *gluing invariants* that ensure the consistency of the system.

One of the key advantages of Event-B is its tooling support. Rodin [6] is an Eclipse based IDE for Event-B that provides effective support for modelling, refinement and proof. Rodin auto-generates POs for project machines. These are then discharged by automated theorem provers, such as AtelierB [2] or SMT [13], or manually via the interactive proving environment. Rodin provides a wide range of plug-ins, such as Camille text editor, statemachine-to-Event-B modelling tool iUML [3] and ProB [17] model checker, which were used in our case study.

2.1 Timing

The Event-B is a general purpose modelling language that lacks explicit support for expressing and verifying timing constraints. However, several concepts were proposed on how to model the time in Event-B. Event-B does not support real numbers natively, hence in this work we discuss only the discrete time related work.

Butler and Falampin [11] describe an approach to model discrete time in Classical B, which is the origin of Event-B. They express current time as a natural number and model the time flow with a tick operation. Deadline conditions are modelled as guards on the tick operation.

Cansell et al. [12] propose a scheme in Event-B. The authors model time as a variable $time \in \mathbb{N}$. An event *post_time* adds a new *active time* to a variable $at \subseteq \mathbb{N}$. Active time elements are the future events' activation times ($min(at) > time$) that must be handled by the system. Event *tick* handles the time flow, where the time progress is limited to the least *at* element – $min(at)$. Event *process_time* then handles the active time. The paper recommends to introduce timing not too early into the model, to avoid unnecessary complexity, especially in terms of proof obligation discharge.

Rehm [20] extends Cansell's work on the active time approach. The author introduces an event-calendar *atCal* that allows to keep a record of the active times for every process. Let *evts* be the finite set of processes or names for one model. Event-calendar is a function that gives for every element of *evts* a set of activation times in the future: $atCal \in evts \rightarrow \mathbb{P}(\mathbb{N})$. In order to facilitate model-checking, [20] shows an approach to refine an infinite model with absolute timing to a finite model with relative timing and show the equivalence of the two models.

Bryans et al. [10], like Rehm, use the extended version of active times, that maps a set of events to future time and adds the support for *bounded inconsistency*. They remove the guard from the *tick* event to allow time to progress beyond the deadline. Instead, they split event *process_time* into two cases. One event then handles the case when the active time is handled within expected time boundaries. The other handles the case when the timing constraint is not correctly maintained by the system.

Sarshogh [21] categorises timing properties in terms of *delay*, *expiry* and *deadline*. He introduces notation to specify these timing properties and provides Event-B semantics for the notation. The notation hides the complexity of encoding timing properties in an Event-B model, thus making timing requirements easier to perceive for the modeller.

In this approach a typical constraint starts with a trigger event followed by a possible response event. A timing constraint relates a trigger event T and a response event R or a set of response events $R_1...R_n$:

$$Deadline(T;\ R_1...R_n;\ t) \tag{2a}$$

$$Delay(T;\ R;\ t) \tag{2b}$$

$$Expiry(T;\ R;\ t) \tag{2c}$$

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

In general, Sarshogh's timing properties correspond to timed automata delay, deadline and time-out modelling patterns [25]. However, two significant differences must be pointed out. Firstly, time in Event-B is modelled explicitly whereas in timed automata it is implicit and continuous ($\mathbb{R}$). Secondly, Sarshogh's patterns can be used in a stepwise refinement modelling, whereas timed automata does not natively support such a feature. We build our approach on Sarshogh's timing properties (2a - 2c) and use similarly structured Event-B semantics.

3 Timing Interval Approach

Our aim is to provide a generative, simple to apply approach to enrich an already existing Event-B model with timing interval constraints. The model can be of any size, may include cyclic and concurrent behaviours and have multiple intervals and other timing constraints.

In the following paragraphs we emphasize the limitations that we solve in our contribution. The need for such timing requirements comes from the pacemaker case study [1] that we have performed [24].

In this paper we present a simple Event-B model [23] to illustrate various modelling needs for timing constraints. The model is abstracted from our pacemaker case study model. We choose a visual state representation for ease of discussion. The abstract model is represented in UML-like diagrams that are generated with the iUML tool as a statemachine *SM* with two concurrent regions (Fig. 1). A transition is enabled when all its source states are active. Therefore $e3$ is always enabled, $e1$ is enabled when the left hand side region is in state A. Transition $e2$ works as a synchronisation point – it is enabled only when the left hand side region is in state st_INT1 and the right hand side region is in state st_INT2. *SM* regions act independently unless the shared event $e2$ is executed.

At the abstract level, we express the timing interval as time spent in a state. In this example we define two intervals $INT1$ and $INT2$ as the time periods during which states st_INT1 and st_INT2 respectively are occupied.

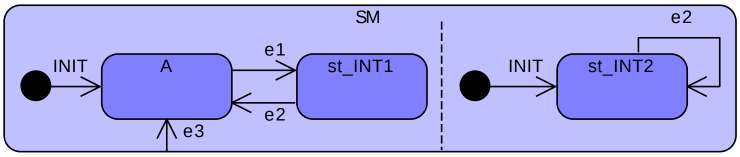

Fig. 1: Example iUML model.

In the left hand side region of the *SM* (Fig. 1), we define an interval $INT1$, triggered by the event $e1$ and responded by the event $e2$. We assume that this interval is an aggregate of delay and deadline timing properties, with lower and upper duration limits. We propose the interval as an abstraction over these properties that formally combine these boundaries. An interval is called *active*, when it has been triggered but not yet responded to.

We consider the notion of *interrupt* event, which can interrupt an already active timing interval. For instance, event $e3$, at any point in time, must be able to interrupt the left hand side region's active timing interval $INT1$. Moreover, we require the enabledness of event $e3$ to be independent of whether $INT1$ is active or not. In contrast, $e2$ is enabled only if there is an active interval to respond to.

The right hand side region contains a timing interval $INT2$. The interval $INT2$ may be triggered by *INIT* or $e2$ event, hence it requires a *multiple trigger support*. Timing interval $INT2$ is responded by the event $e2$.

Note that both timing intervals are interdependent – they share the event $e2$, effectively forcing a single event to serve as a response for the $INT1$ and as both trigger and response for $INT2$. We call this phenomenon *event overloading*, when an event serves a number of roles in one or more timing intervals.

3.1 Modelling Notation

In order to model the given example, we introduce the timing interval approach. The interval is characterised by one or two *timing properties TP* and a set of events – optional ones denoted by [ ]. The system may have a number of timing intervals that are identified by a unique name – *Interval_name*. There may be multiple active instances of a given interval that occur independently from each other.

$$Interval\_name(T_1[,...,T_i];\ R_1[,...,R_j];\ [I_1,...,I_k];\ TP_1(t_1)[,TP_2(t_2)]) \qquad (3)$$

The interval is defined by three kinds of events. One of a set of trigger events $T \in T_1..T_i$ always creates a new instance of the interval. One of a set of response events $R \in R_1...R_j$ always terminates 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 response event. One of a set of optional interrupt events $I \in I_1..I_k$ interrupts the interval. Unlike the response event, the interrupt event is not constrained by timing properties TP and does not block if there is no active interval instance to interrupt. The interrupt event always interrupts 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*, *Delay* or *Expiry*. Further, the interval can have one of five TP configurations: (i.) Deadline; (ii.) Delay; (iii.) Expiry; (iv.) Delay and Expiry; (v.) Delay and Deadline. If more than one timing property is associated with an interval, then there is a relation between the interval's timing property durations (2a-2c): the delay duration must be less or equal to the deadline duration ($t_{Delay} \leq t_{Deadline}$) and the expiry duration ($t_{Delay} \leq t_{Expiry}$).

Having defined the notation, we can now use it to specify the left hand side region timing constraint $INT1$ ((4), Fig. 1), with trigger $e1$, response $e2$ and interrupt $e3$. Upon event $e1$ execution, a new interval $INT1$ instance is created. The occurrence of the response event $e2$ then becomes constrained by the delay and deadline timing properties whose durations are $INT1\_t\_dly$ and $INT1\_t\_ddl$ respectively. The interrupt event $e3$ can be executed at any given time regardless of the state the model is in. Upon event $e3$ execution the active $INT1$ instance is interrupted (if one exists) and the left hand side region enters state A.

$$INT1(e1;\ e2;\ e3;\ Delay(INT1\_t\_dly), Deadline(INT1\_t\_ddl)) \qquad (4)$$

According to the interval $INT2$ specification (5), the right hand side (Fig. 1) interval is triggered by $INIT$ or $e2$ events. Event $INIT$ means that the interval is activated immediately upon the model initialisation. The *overloaded* $e2$ event serves as the trigger and the response for the interval $INT2$. Therefore when executed, event $e2$ responds to an already existing interval instance and initiates a new one. The deadline timing property means that event $e2$ must occur within time $INT2\_t\_ddl$ of trigger event occurring. $INT2$ has no interrupt and therefore can be responded to only by the response event $e2$.

$$INT2(INIT, e2;\ e2;\ ;\ Deadline(INT2\_t\_ddl)) \tag{5}$$

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

3.2 Semantics of Example Intervals

We give semantics to our interval construct by translating it to Event-B variables, invariants, guards and actions. The interval timing notation serves as a blueprint, indicating the required Event-B code and its location in the model. In this section we provide semantics of the example interval $INT1$.

Interval. We translate the interval $INT1$ to a set of variables that store the information about interval instances (Fig. 2). Variable $INT1\_trig$ stores the indices of triggered interval $INT1$ instances. When the interval instance is responded to, its index is copied to the $INT1\_resp$ variable. Trigger and response activities are timestamped and the timestamps are stored in $INT1\_trig\_ts$ and $INT1\_resp\_ts$ variables respectively. We model timestamp as a total function $X \rightarrow \mathbb{N}$, where the index set X serves as a unique identification for the interval instance. In case the interval is interrupted, its index is copied to variable $INT1\_intr$. Interval $INT1$-specific variables are prefixed with $INT1\_$.

Invariants $INT1\_consist_1$ and $INT1\_consist_2$ ensure the interval index consistency across the variables (Fig. 2). $INT1\_consist_1$ is the sequencing invariant ensuring that only the triggered indexes can be responded to or interrupted. $INT1\_consist_2$ states that interval instance can be either responded to or interrupted, but not both.

Timing Properties. In Event-B semantics, the timing property is expressed as a set of invariants (Fig. 7). According to $INT1$ specification (4), the interval is constrained by two timing properties: the delay and the deadline. The deadline timing property consists of two invariants. The first invariant $INT1\_inv\_ddl1$ expresses the requirement, that while the active interval instance has not yet been responded to or interrupted, it must not exceed the deadline duration $INT1\_t\_ddl$. The second deadline invariant $INT1\_inv\_ddl2$ requires the active interval $INT1$ instance to be responded to within $INT1\_t\_ddl$ of the trigger event occurring. In order to preserve $INT1$ deadline timing property invariants, a guard $INT1\_grd\_ddl1$ is needed in the *tick* event to ensure that the time will not progress beyond active interval's deadline boundaries (Fig. 6).

The delay timing property of $INT1$ is expressed as one invariant $INT1\_inv\_dly1$ (Fig. 7). The guard $INT1\_grd\_dly1$ in event $e2$ ensures the invariant preservation (Fig. 4). Note that event *tick* (Fig. 6) is not constrained by delay timing properties.

```
INT1_type1 :  INT1_trig ⊆ X
INT1_type2 :  INT1_resp ⊆ X
INT1_type3 :  INT1_intr ⊆ X
INT1_type4 :  INT1_trig_ts ∈ INT1_trig → ℕ
INT1_type5 :  INT1_resp_ts ∈ INT1_resp → ℕ
INT1_consist1 :  ∀ idx · idx ∉ INT1_trig
                ⇒ idx ∉ INT1_resp ∪ INT1_intr

INT1_consist2 :  INT1_intr ∩ INT1_resp = ∅
```

Fig. 2: Interval $INT1$ variables.

```
Event   e1 ≙
 any  INT1_pTrig
 where
 Grds
 INT1_trg_grd1 :  INT1_pTrig ∈ X
 INT1_trg_grd2 :  INT1_pTrig ∉ INT1_trig
 then
 Acts
 INT1_trg_act1 :
   INT1_trig := INT1_trig ∪ {INT1_pTrig}
 INT1_trg_act2 :
   INT1_trig_ts(INT1_pTrig) := time
 end
```

Fig. 3: Event $e1$.

```
Event   e2 ≙
 any  INT1_pResp INT2_pTrig INT2_pResp
 where
 Grds
 INT1_rsp_grd1 :  INT1_pResp ∈ INT1_trig
 INT1_rsp_grd2 :  INT1_pResp ∉ INT1_resp ∪ INT1_intr
 INT1_grd_dly1 :  time ≥ INT1_trig_ts(INT1_pResp) + INT1_t_dly
 INT2_trg_grd1 :  INT2_pTrig ∈ X
 INT2_trg_grd2 :  INT2_pTrig ∉ INT2_trig
 INT2_rsp_grd1 :  INT2_pResp ∈ INT2_trig
 INT2_rsp_grd2 :  INT2_pResp ∉ INT2_resp ∪ INT2_intr
 then
 Acts
 INT1_rsp_act1 :  INT1_resp := INT1_resp ∪ {INT1_pResp}
 INT1_rsp_act2 :  INT1_resp_ts(INT1_pResp) := time
 INT2_trg_act1 :  INT2_trig := INT2_trig ∪ {INT2_pTrig}
 INT2_trg_act2 :  INT2_trig_ts(INT2_pTrig) := time
 INT2_rsp_act1 :  INT2_resp := INT2_resp ∪ {INT2_pResp}
 INT2_rsp_act2 :  INT2_resp_ts(INT2_pResp) := time
 end
```

Fig. 4: Event $e2$.

```
Event   e3 ≙
 any  INT1_pIntr
 where
 Grds
 INT1_intr_grd1 :  INT1_pIntr ⊆ INT1_trig \ (INT1_resp ∪ INT1_intr)
 INT1_intr_grd2 :  finite(INT1_pIntr)
 INT1_intr_grd3 :  INT1_trig \ (INT1_resp ∪ INT1_intr) ≠ ∅ ⇒ card(INT1_pIntr) = 1
 then
 Acts
 INT1_intr_act1 :  INT1_intr := INT1_intr ∪ INT1_pIntr
 end
```

Fig. 5: Event $e3$.

```
Event   tick ≙
 when
 Grds
 INT1_grd_ddl1 :  ∀ idx·idx ∈ INT1_trig ∧ idx ∉ INT1_resp ∪ INT1_intr
                              ⇒time + 1 ≤ INT1_trig_ts(idx) + INT1_t_ddl
 INT2_grd_ddl1 :  ∀ idx·idx ∈ INT2_trig ∧ idx ∉ INT2_resp ∪ INT2_intr
                              ⇒time + 1 ≤ INT2_trig_ts(idx) + INT2_t_ddl
 then
 Acts
 act1 :  time := time + 1
 end
```

Fig. 6: *tick* event.

```
INT1_inv_ddl1 :  ∀ idx·idx ∈ INT1_trig ∧ idx ∉ INT1_resp ∪ INT1_intr
                             ⇒time ≤ INT1_trig_ts(idx) + INT1_t_ddl
INT1_inv_ddl2 :  ∀ idx·idx ∈ INT1_trig ∧ idx ∈ INT1_resp
                             ⇒INT1_resp_ts(idx) ≤ INT1_trig_ts(idx) + INT1_t_ddl
INT1_inv_dly1 :  ∀ idx·idx ∈ INT1_trig ∧ idx ∈ INT1_resp
                             ⇒INT1_resp_ts(idx) ≥ INT1_trig_ts(idx) + INT1_t_dly
INT1_rel_dly_ddl :  INT1_t_dly ≤ INT1_t_ddl
```

Fig. 7: Interval $INT1$ timing property invariants.

Invariant $INT1\_rel\_dly\_ddl$ (Fig. 7) specifies the relation between delay and deadline timing property durations.

Events. According to the $INV1$ specification (4), event $e1$ serves as the trigger for $INT1$ (Fig. 3). To trigger a new instance of the interval, event accepts a parameter $INT1\_pTrig$ that must be an unused index ($INT1\_trg\_grd1$, $INT1\_trg\_grd2$). If the conditions are met, the new index and the timestamp are added to $INT1$ trigger and timestamp sets ($INT1\_trg\_act1$, $INT1\_trg\_act2$). Event $e2$ serves as $INT1$ response (Fig. 4). $e2$ takes a parameter $INT1\_pResp$ that must be an already existing interval $INT1$ index and has not yet been responded to or interrupted ($INT1\_rsp\_grd1$, $INT1\_rsp\_grd2$). Upon response, the selected index is recorded into the responded event set $INT1\_resp$ with its timestamp ($INT1\_rsp\_act1$, $INT1\_rsp\_act2$). *Grds* represents the other guards and *Acts* represents the other actions of the corresponding event.

Event $e2$ is an example of an overloaded event. It serves as the response for $INT1$ and as both, trigger and response for $INT2$ (Fig. 4). $INT2$ trigger parameter $INT2\_pTrig$ and labels $INT2\_trg\_*$ correspond to those of $INT1$; In an analogous manner, $INT2$ response parameter $INT2\_pResp$ and labels $INT2\_rsp\_*$ match the ones of $INT1$. As mentioned in subsection 3.1, the response event must always respond to an active interval instance. Hence $e2$ can be executed only when there are active instances of intervals $INT1$ and $INT2$ to respond to, otherwise the event is disabled. There is no interference between these three roles, as they operate on different variables.

Event $e3$ serves as an interrupt for $INT1$ (Fig. 5). Parameter $INT1\_pIntr$ is modelled as a subset of active but non-responded $INT1$ instance indexes ($INT1\_grd1$). If, upon event execution, there is no active $INT1$ instance, the parameter becomes equal to $\varnothing$ and interval's variable is not affected ($INT1\_act1$). On the other hand, if there is at least one active interval instance available, the parameter is forced to contain one index ($INT1\_grd3$). The guard $INT1\_grd2$ is required for well-definedness, since *cardinality* function can accept only finite parameters. We limit $INT1\_pIntr$ size to 1 to ensure a consistent behaviour with trigger and response parameters that always accept strictly one element.

4 Interval Templates

We provide a generative approach for translating interval specification to Event-B code. Our approach defines a number of generic Event-B code templates that represent elements of the interval notation (3). The templates can potentially be specialised, and thus simplified, to handle, e.g., strictly a single instance interval.

The interval timing approach consists of the *interval base template, event templates* and *timing property templates.* Our process comprises three steps. Firstly, we pick relevant templates according to the interval specification. Then, we instantiate the templates by adding the interval name as a prefix to each template variable (as for $INT1\_$ and $INT2\_$ prefixes in the previous sections). Finally, we inject instantiated templates into the model locations, specified by the interval specification.

Interval Base Template. The interval base template is a set of variables and invariants that describe all interval instance states and ensures their consistency (Fig. 8). Prefix **P_** is a place holder for the interval name that gets instantiated in the template code. **@** indicates the target Event-B block to be injected with the instantiated template code.

```
@INVARIANTS
P_type1 :  P_trig ⊆ X                      P_consist1 :  ∀idx·idx ∉ P_trig ⇒ idx ∉ P_resp ∪
P_type2 :  P_resp ⊆ X                            P_intr
P_type3 :  P_intr ⊆ X                      P_consist2 :  P_intr ∩ P_resp = ∅
P_type4 :  P_trig_ts ∈ P_trig → ℕ
P_type5 :  P_resp_ts ∈ P_resp → ℕ
```

Fig. 8: Interval base template elements.

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

The deadline timing property template consists of two invariants and a guard in *Tick* event (Fig. 9). Invariants $\mathbf{P\_}\mathit{inv\_ddl}1$ and $\mathbf{P\_}\mathit{inv\_ddl}2$ expresses the deadline timing property requirement. Guard $\mathbf{P\_}\mathit{grd\_ddl}1$ is for *Tick* event.[1]

```
@INVARIANTS
P_inv_ddl1 :  ∀idx·idx ∈ P_trig ∧ idx ∉ P_resp ∪ P_intr ⇒ time ≤ P_trig_ts(idx) + P_t_ddl
P_inv_ddl2 :  ∀idx·idx ∈ P_trig ∧ idx ∈ P_resp ⇒ P_resp_ts(idx) ≤ P_trig_ts(idx) + P_t_ddl
@Event    Tick ≙
@where
P_grd_ddl1 :  ∀idx·idx ∈ P_trig ∧ idx ∉ P_resp ∪ P_intr ⇒ time + tick ≤ P_trig_ts(idx) +
P_t_ddl
end
```

Fig. 9: Deadline template.

The delay timing property template consists of a single invariant $\mathbf{P\_}\mathit{inv\_dly}1$ and a guard $\mathbf{P\_}\mathit{grd\_dly}1$ on a response event (Fig. 10).

```
P_inv_dly1 :  ∀idx·idx ∈ P_trig ∧ idx ∈ P_resp⇒P_resp_ts(idx) ≥ P_trig_ts(idx)+P_t_dly
@Event    R ≙
@where
P_grd_dly1 :  time ≥ P_trig_ts(P_pResp) + P_t_dly
end
```

Fig. 10: Delay template.

In case interval has delay and deadline (Fig. 11) or delay and expiry (Fig. 12) timing properties, their duration relation is specified as an invariant.

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

Interval Event Templates. We define Event-B code templates for trigger T (Fig. 13), response R (Fig. 14) and interrupt I (Fig. 15) interval event types. Templates consist of parameters, guards and actions that are needed for a specific interval role. The templates are analogous to $INT1$ trigger (Fig. 3), response (Fig. 4) and interrupt (Fig. 5).

```
@INVARIANTS
P_rel_dly_ddl : P_t_dly ≤ P_t_ddl
```

Fig. 11: Delay-deadline TP rel. tl.

```
@INVARIANTS
P_rel_dly_xpr : P_t_dly ≤ P_t_xpr
```

Fig. 12: Delay-expiry TP rel. tl.

```
@Event   T ≙
@any  P_pTrig
@where
P_trg_grd1 :  P_pTrig ∈ X
P_trg_grd2 :  P_pTrig ∉ P_trig
@then
P_trg_act1 :  P_trig := P_trig ∪ {P_pTrig}
P_trg_act2 :  P_trig_ts(P_pTrig) := time
end
```

Fig. 13: Trigger event template.

```
@Event   R ≙
@any  P_pResp
@where
P_rsp_grd1 :  P_pResp ∈ P_trig
P_rsp_grd2 :  P_pResp ∉ P_resp ∪ P_intr
@then
P_rsp_act1 :  P_resp := P_resp ∪ {P_pResp}
P_rsp_act2 :  P_resp_ts(P_pResp) := time
end
```

Fig. 14: Response event template.

```
@Event   I ≙
@any  P_pIntr
@where
P_intr_grd1 :  P_pIntr  ⊆  P_trig \ (P_resp ∪
     P_intr)
P_intr_grd2 :  finite(P_pIntr)
P_intr_grd3 :  P_trig \ (P_resp ∪ P_intr) ≠ ∅ ⇒
          card(P_pIntr) = 1
@then
P_intr_act1 :  P_intr := P_intr ∪ P_pIntr
end
```

Fig. 15: Interrupt event template.

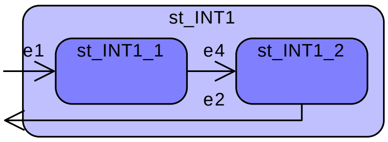

Fig. 16: Ref. of PM example.

5 Example Interval Refinement to Sequential Sub-Intervals

We chose one interval refinement pattern out of a number of possible ones [21]. In this section we demonstrate in our example model how the abstract timing interval $INT1$ (4) can be refined into two sub-intervals $INT1\_1$ (6) and $INT1\_2$ (7). We visually express sub-intervals as sub-states st_INT1_1 and st_INT1_2 of the parent state st_INT1 (Fig. 16). Sub-states are connected with a new transition $e4$.

Sub-intervals are modelled in the same way as the abstract interval $INT1$ unless stated otherwise. Concrete sub-intervals $INT1\_1$ and $INT1\_2$ have their own trigger, response and interrupt variables, and at least the same number and type of timing properties. Concrete sub-intervals proceed sequentially, where preceding interval's response serves as succeeding interval's trigger. Thus the

$INT1\_1$ response $e4$ serves as the trigger for $INT1\_2$.

$$INT1\_1(e1;\ e4;\ e3;\ Delay(INT1\_1\_t\_dly), Deadline(INT1\_1\_t\_ddl)) \tag{6}$$

$$INT1\_2(e4;\ e2;\ e3;\ Delay(INT1\_2\_t\_dly), Deadline(INT1\_2\_t\_ddl)) \tag{7}$$

This interval refinement is encoded by a set of gluing invariants that map abstract interval variables to concrete sub-interval variables.

Firstly, the concrete sub-interval $INT1\_1$ must data refine all abstract interval $INT1$ trigger variables. Interval $INT1$ trigger index and trigger timestamp variables must map to interval $INT1\_1$ trigger index and timestamp (8). Secondly, abstract interval $INT1$'s response index variables must be refined (9).

$$INT1\_trig = INT1\_1\_trig \wedge INT1\_trig\_ts = INT1\_1\_trig\_ts \tag{8}$$

$$INT1\_resp = INT1\_2\_resp \wedge INT1\_resp\_ts = INT1\_resp\_ts \tag{9}$$

Thirdly, $INT1$'s interrupt indexes must be refined (10).The concrete interrupt indices must be unique to each sub-interval.

$$INT1\_intr = INT1\_1\_intr \cup INT1\_2\_intr \wedge INT1\_1\_intr \cap INT1\_2\_intr = \varnothing \tag{10}$$

Note that in the refined model of subsection 3.2, event $e3$ acts as interrupt for both $INT1\_1$ and $INT1\_2$ intervals (Fig. 17). We reuse the interrupt event pattern. In case there are no active interval instances, both interrupt index parameters become empty sets. Otherwise, guards $INT1\_1\_intr\_grd3$ and $INT1\_2\_intr\_grd3$ force strictly one of the parameters to be a non empty set with the cardinality of 1. The *with* Event-B keyword (*witness*) defines the relation between the abstract parameter that has been refined away and concrete parameters. In event $e3$ witness $INT1\_pIntr$ specifies that the indexes of interrupted abstract and concrete intervals must match.

```
Event   e3 ≙
 refines  e3
 any  INT1_1_pIntr INT1_2_pIntr
 where
 seq_grd :  SM = TRUE
 INT1_1_intr_grd1 :  INT1_1_pIntr ⊆ INT1_1_trig \ (INT1_1_resp ∪ INT1_1_intr)
 INT1_2_intr_grd1 :  INT1_2_pIntr ⊆ INT1_2_trig \ (INT1_2_resp ∪ INT1_2_intr)
 INT1_1_intr_grd2 :  finite(INT1_1_pIntr)
 INT1_2_intr_grd2 :  finite(INT1_2_pIntr)
 INT1_1_intr_grd3 :  INT1_1_trig  \  (INT1_1_resp  ∪  INT1_1_intr)   ≠   ∅  ⇒  card(INT1_1_pIntr  ∪
 INT1_2_pIntr) = 1
 INT1_2_intr_grd3 :  INT1_2_trig  \  (INT1_2_resp  ∪  INT1_2_intr)   ≠   ∅  ⇒  card(INT1_1_pIntr  ∪
 INT1_2_pIntr) = 1
 with
 INT1_pIntr :  INT1_pIntr = INT1_1_pIntr ∪ INT1_2_pIntr
 then
 seq_act :  C := TRUE, A := TRUE, B := FALSE, B2 := FALSE, B1 := FALSE
 INT1_1_intr_act1 :  INT1_1_intr := INT1_1_intr ∪ INT1_1_pIntr
 INT1_2_intr_act2 :  INT1_2_intr := INT1_2_intr ∪ INT1_2_pIntr
 end
```

Fig. 17: m1: refined interrupt event $e3$.

Finally, interval $INT1\_1$ response indexes and timestamps must map to interval $INT1\_2$ indexes and timestamps (11). This ensures the continuity of concrete intervals.

$$INT1\_1\_resp = INT1\_2\_trig \wedge INT1\_1\_resp\_ts = INT1\_2\_trig\_ts \tag{11}$$

To make sure that concrete sub-intervals do not violate abstract interval durations, the relation between timing property durations is specified as invariants. The sum of sub-interval deadline property durations must be less or equal to the abstract interval's deadline property duration (12). The sum of sub-interval delay property durations must be higher or equal to abstract interval's delay property duration (13).

$$INT1\_1\_t\_ddl + INT1\_2\_t\_ddl \leq INT1\_t\_ddl \tag{12}$$

$$INT1\_1\_t\_dly + INT1\_2\_t\_dly \geq INT1\_t\_dly \tag{13}$$

6 Verification and Validation

We have evaluated our timing interval approach in terms of applicability, verification and validation. The refinement model has 3 timing intervals ($INT1\_1$, $INT1\_2$ and $INT2$) and 47 time-related invariants. All 132 generated timing-related POs were automatically discharged. Verification for deadlock freeness is not well integrated into Event-B framework [26], hence we favour model-checking for this task. To further verify our approach, we have model-checked our model with a limited state-space coverage and did not find any deadlocks of invariant violations. Since we model time as an absolute value of $\mathbb{N}$, the infinite state space prevents us from a full state-space coverage. Finally, the model has been manually animated in the ProB and there were no invariant violations or deadlocks found.

A fuller evaluation of our approach is the pacemaker case study [24]. The pacemaker model resulted in three refinements with the final refinement having 10 timing intervals. No customisations were needed to our approach in order to model the timing requirements. Overall, the model has 177 timing related invariants. There are 652 time-related proof obligations, all of which were automatically discharged. A limited coverage model checking has been performed using ProB model-checker. No deadlocks or invariant violations were found, so our approach appears to scale.

We have written a number of test case scenarios for manual validation with the ProB animator in order to test various aspects of the model and the timing interval approach.

Finally, we have developed a heart model in Groovy language for ProB model checker [9]. The heart model has been written as a Java plug-in. It is a simplistic system with two methods *isVentricleContracted*() and *isAtriumContracted*() that return a random boolean value. The simulation engine performs actions in a sequential loop fashion: (i.) invokes the methods to update the heart model state (ii.) if appropriate, executes pacemaker model *sense* events (iii.) arbitrarily executes any non-*sense* pacemaker model event. The simulation did not return any negative results.

7 Related Work

A number of authors have modelled the pacemaker system. Each case study differs in the covered scope of requirements and the modelling challenges that authors have perceived and tackled. For timing, we note some modelling improvement our approach offers over other work.

[19] have developed a single electrode pacemaker system using Event-B. The authors used the *activation times* pattern [12] to model timing constraints. They did not treat timing constraints as a separate element but rather integrated them tightly into the model. Timing constraint implementation is tightly coupled with the model structure, thus does not take advantage of reusability and requires more modelling effort. [16] used timed automata to model a closed loop system of the two-channel pacemaker and the heart. Since UPPAAL lacks a notion of refinement, the complexity of the system is put all at once in a component oriented fashion. The authors modelled pacemaker timing intervals as separate automata that correspond to time counters. The automata communicate via broadcast channels. This is a more complex bottom-up approach than ours. Other works include [18], [14] and [15]. None of the reviewed case studies uses notation specific to timing requirements.

We have chosen to model a dual-channel pacemaker. The support of refinement in Event-B allowed us to use a top-down approach, dealing with the system complexity incrementally. We have expressed the pacemaker system as two interdependent statemachines, representing atrium and ventricle channels. Interdependency and concurrent behaviour are the main factors for the complexity of our the model. To specify the requirements, we used the timing interval notation. We then generated explicit time constraints using our approach, that required no customisations.

8 Conclusions and Future Work

In the simple example model we have highlighted some timing aspects of a complex critical system and demonstrated how to overcome them using our approach. From the case study results we have concluded that the introduced notation gives a sufficient degree of flexibility in terms of timing requirement specification. The example model shows how the interrupt event facilitates event interruption by non-deterministic events and helps to avoid event replication to tackle different cases. As demonstrated in the example model, the event can be overloaded, that is, serve many event roles (trigger, response or interrupt) for multiple intervals. Our approach decouples intervals from other model structure. This affords a template-driven generative approach to modelling timing.

We plan to formalise the interval refinement of section 5 and provide templates for generative modelling. Further, we plan to present more refinement patterns [21].

Two factors prevent the full state-space coverage model-checking. Firstly, we model time as absolute value $\mathbb{N}$. Secondly, the interval instance indexes are not discarded after the use and accumulate. To overcome the infinite state-space problem we consider introducing a relative countdown timer for modelling cyclic intervals [20] and an index reset method for our approach that clears used interval instance indices.

More complex pacemaker systems support variable timing intervals, therefore in future we plan to implement a variable duration t for timing properties. We intend to use a co-simulation plug-in [22] to validate our model against more sophisticated heart models.

Finally, our plan is to develop a plug-in for Event-B code generation, add visualisation support for timing interval representation in iUML diagrams and ProB animations.

Bibliography

[1] Pacemaker Challenge, http://sqrl.mcmaster.ca/pacemaker.htm, 2007.

[2] Interactive Prover Reference Manual 3.7, http://www.atelierb.eu/ressources/DOC/english/prover-reference-manual.pdf, 2013.

[3] iUML, http://wiki.event-b.org/index.php/IUML-B, 2013.

[4] J.-R. Abrial. *The B-Book: Assigning Programs to Meanings.* Cambridge University Press, New York, NY, USA, 1996.

[5] J.-R. Abrial. *Modeling in Event-B: System and Software Engineering.* Cambridge University Press, New York, NY, USA, 1st edition, 2010.

[6] J.-R. Abrial, M. Butler, S. Hallerstede, T. S. Hoang, F. Mehta, and L. Voisin. Rodin: an Open Toolset for Modelling and Reasoning in Event-B. *International Journal on Software Tools for Technology Transfer*, 12(6):447–466, Nov. 2010.

[7] R.-J. Back and R. Kurki-Suonio. Decentralization of Process Nets with Centralized Control. In *Symposium on Principles of Distributed Computing*, pages 131–142, Montreal, Quebec, Canada, 1983. ACM.

[8] S. S. Barold, R. Stroobandt, and A. F. Sinnaeve. *Cardiac Pacemakers and Resynchronization Step-by-Step: an Illustrated Guide.* Wiley-Blackwell, 2010.

[9] J. Bendisposto. ProB 2.0 Developer Handbook, http://nightly.cobra.cs.uni-duesseldorf.de/prob2/developer-documentation/prob-devel.pdf, 2014.

[10] J. Bryans, J. Fitzgerald, A. Romanovsky, and A. Roth. Patterns for Modelling Time and Consistency in Business Information Systems. pages 105–114, Oxford, UK, Mar. 2010. IEEE Computer Society.

[11] M. Butler and J. Falampin. An Approach to Modelling and Refining Timing Properties in B. In *Proceedings of Workshop on Refinement of Critical Systems (RCS)*, Jan. 2002.

[12] D. Cansell, D. Méry, and J. Rehm. Time Constraint Patterns for Event B Development. In *B 2007: Formal Specification and Development in B*, volume 4355 of *LNCS*, pages 140–154. Springer, 2006.

[13] D. Déharbe, P. Fontaine, Y. Guyot, and L. Voisin. SMT Solvers for Rodin. In *Abstract State Machines, Alloy, B, VDM, and Z*, volume 7316 of *LNCS*, pages 194–207. Springer, 2012.

[14] A. O. Gomes and M. Oliveira. Formal Development of a Cardiac Pacemaker: From Specification to Code. In *Formal Methods: Foundations and Applications*, volume 6527 of *LNCS*, pages 210–225. Springer, 2011.

[15] E. Jee, S. Wang, J. K. Kim, J. Lee, O. Sokolsky, and I. Lee. A Safety-Assured Development Approach for Real-Time Software. In *The Proceedings of the 16th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 133–142, Aug. 2010.

[16] Z. Jiang, M. Pajic, S. Moarref, R. Alur, and R. Mangharam. Modeling and Verification of a Dual Chamber Implantable Pacemaker. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 7214 of *LNCS*, pages 188–203. Springer, 2012.

[17] M. Leuschel and M. Butler. ProB: A Model Checker for B. In *FME 2003: Formal Methods*, volume 2805 of *LNCS*, pages 855–874. Springer, 2003.

[18] H. Macedo, P. Larsen, and J. Fitzgerald. Incremental Development of a Distributed Real-Time Model of a Cardiac Pacing System Using VDM. In *FM 2008: Formal Methods*, volume 5014 of *LNCS*, pages 181–197. Springer, 2008.

[19] D. Méry and N. K. Singh. Pacemaker's Functional Behaviors in Event-B. Research Report inria-00419973, 2009.

[20] J. Rehm. From Absolute-Timer to Relative-Countdown: Patterns for Model-Checking. Unpublished. May 2008.

[21] M. R. Sarshogh. *Extending Event-B with Discrete Timing Properties*. PhD thesis, University of Southampton, 2013.

[22] V. Savicks, M. Butler, and J. Colley. Co-simulating Event-B and Continuous Models via FMI. In *2014 Summer Computer Simulation Conference*. Society for Modeling & Simulation International (SCS), July 2014.

[23] G. Sulskus, M. Poppleton, and A. Rezazadeh. Example Event-B project, http://users.ecs.soton.ac.uk/gs6g10/SimplifiedPMExample.zip, 2014.

[24] G. Sulskus, M. Poppleton, and A. Rezazadeh. *An Investigation into Event-B Methodologies and Timing Constraint Modelling*. Mini-Thesis, University of Southampton, 2014.

[25] J. Wang. *Handbook of Finite State Based Models and Applications*. Discrete Mathematics and Its Applications. Chapman And Hall/CRC, 2012.

[26] F. Yang and J.-P. Jacquot. Scaling Up with Event-B: A Case Study. In *NASA Formal Methods*, volume 6617 of *LNCS*, pages 438–452. Springer, 2011.