Abstract. A large number of dependable embedded systems have stringent real-time requirements imposed on them. An analysis of real-time behaviour is usually conducted at the implementation level. However, it is desirable to obtain an evaluation of real-time properties early at the development cycle, i.e., at the modelling stage. In this paper we present an approach to augmenting Event B modelling with verification of real-time properties in Uppaal. Event B provides the developers with a scalable approach to modelling and verification of functional requirements. We show how to extract a process-based view from an Event B model that together with introducing time constraints allows us to obtain timed automata model – the input model of Uppaal. The approach is illustrated by an excerpt from a realistic case study – data processing on-board software of BePi Colombo Mission.

1 Introduction

Event B [1] offers a scalable approach to correct-by-construction system development. While developing system in Event B we start from an abstract system model that represents only the most essential system behaviour and properties. In a number of correctness-preserving model transformations – refinements – we arrive at a sufficiently detailed system specification. Each refinement step is accompanied by proofs.

The focus of Event B modelling is functional requirements. However, in the design of embedded systems, non-functional requirements, such as real-time, play equally important role. Usually, real-time systems characteristics are evaluated at the late development stages, i.e., at the implementation level. It might incur costly redevelopment, if the real-time constraints are not met. Hence, it is desirable to integrate modelling of functional and non-functional requirements to obtain early evaluation of system real-time properties.

To enable verification of real-time, several approaches relied on introducing an explicit notion (variable) time into the Event B model [2–6]. However, such a technique leads to cluttering a model and suffers from a poor scalability.

In this paper we propose a different approach. It is based on extracting an auxiliary model – a Process View model – from Event B representation of system functionality. While suppressing the details of functional system behaviour, the Process View model provides the explicit notions of process and synchronization. We apply the rely/guarantee reasoning [7] to define process abstractions in
a compositional way. The Uppaal model checker [8] is used to verify liveness and real-time properties of Process View models augmented with clocks and time constraints. An important concern is the verification of Process View consistency and the link between a Process View model and Event-B machine. All the relevant conditions are expressed in the way that makes them amenable to the verification within the Rodin Toolkit [9] - an environment for the development of Event-B models.

Our approach is illustrated by an excerpt from an industrial case study – development of data processing unit of on-board software of BePi Colombo satellite undertaken within EU FP7 Project Deploy [10]. The initial development was undertaken by the company Space Systems Finland. The first modelling attempt focused on capturing functional requirements only. However, even though the system is large and complex it has rather simple safety properties. The approach, which we will present in this paper, is our first successful attempt to find a scalable solution to verify intricate liveness and real-time properties of a large industrial-size system.

Our approach aims at facilitating investigation of real-time behaviour at the modelling stage rather than replacing powerful simulation techniques for analysing real-time system characteristics at the implementation level. It helps the designers to explore the impact of various architectural alternatives on real-time and hence makes the development process more predictable.

2 Background

The B Method [11] is an approach for the industrial development of highly dependable software. The method has been successfully used in the development of several complex real-life applications [12,13]. Event-B [14] is a formal framework derived from the B Method to model parallel, distributed and reactive systems. The Rodin platform [9] provides automated tool support for modelling and verification in Event-B. Currently Event-B is used in the EU project Deploy [15] to model industrial systems from space, automotive, railway, and business domains.

In Event-B, a system specification (model) is an abstract state machine [1]. An abstract state machine encapsulates the model state, represented as a collection of model variables, and defines operations on this state. A machine may also have the accompanying component, called context, containing user-defined carrier sets, constants, and their properties given as a list of model axioms.

The dynamic behaviour of the system is defined by the set of atomic events specified in the Events clause. Generally, an event can be defined as follows:

\[ \text{evt} \triangleq \text{any } u \text{ where } g \text{ then } S \text{ end}, \]

where \( u \) is a list of local event variables, the guard \( g \) is a conjunction of predicates over the global \((v)\) and local \((u)\) variables, and the action \( S \) is an assignment to the state variables.

Event-B employs a top-down refinement-based approach to system development. Development starts from an abstract system specification that models the
most essential functional requirements. While capturing more detailed requirements, each refinement step typically introduces new events and variables into the abstract specification. Moreover, Event-B supports data refinement, allowing us to replace some abstract variables with their concrete counterparts.

Consistency of models as well as correctness of refinement steps are expressed as a number of proof obligations. The model verification effort, in particular, automatic generation and proving of the required proof obligations, is significantly facilitated by the provided tool support – the Event B platform.

Uppaal \cite{8} is a tool for animation and verification of timed automata \cite{16} - a modelling language for the specification and analysis of real-time systems. Timed automaton is a finite automaton with the notion of dense time defined by a number of real-valued clocks. The state of such automaton is defined by its current location and the values of all the clocks. The progress of time affects the behaviour of an automaton. To capture this, edges are annotated with constraints on clock values. Such constraints denote deadlines for the corresponding transitions. Clock constraints may also be placed in locations. These are used to ensure the progress in the system with the flow of time. All clocks progress synchronously but each clock may be reset independently of other clocks.

3 Process View

Event B does not offer model structuring mechanisms beyond that of event (an atomic state transition). Regardless of the implied purpose and the structure of a modelled system, a model appears as a flat list of events. Structuring constructs found in other formalisms we deliberately excluded from Event B. Instead, it was decided to have a simple core language with excellent automated proof rate and let the formal methods community experiment with method and notation extensions. Supporting this idea there is an open (in sense of extensibility) Event B toolset called Rodin Platform.

It is our opinion that a major obstacle to the verification of timing properties of Event B is an absence of a clear picture of concurrency. The Event B model of concurrency is maximum concurrency where a possible simulation of a model is running as many events as possible concurrently. This leads to a confusing picture where the number of threads of control change constantly and there are no simple notions of process and synchronisation. The approach we have chosen is to build, independently of Event B, a model with an explicit notion of a process. This lets us use the Event B language and its refinement method but we have to provide a link between this new kind of model and an Event B model for which process abstraction is defined. The practicables of the approach are important and we aim to mechanise the described method in the Rodin Platform. For this reason verification conditions are expressed as FOL theorems.

The rest of the section is organised as follows. We introduce the basic building blocks of a process - activity and activity transitions. We define how to assemble activities into a process and reason at the level of a process. Finally, we give a
definition of a system of communicating processes and apply the rely/guarantee reasoning to establish process compatibility.

3.1 Activity and Activity Transition

The simplest form of a process is called an activity - an abstract characterisation of a piece of functionality. An encapsulated functionality may be anything ranging from a single assignment to a complex of software and hardware components. Crucial here is that the execution of an activity is not atomic: an activity takes certain time to complete, while other processes may also evolve in the meantime.

As a part of Process View (PV) model, an activity is a black-box. However, when a link is made to an Event B model, this black-box is opened. Such temporary hiding of details is the cornerstone of our approach. It is necessary to ignore for a while some details of an Event B model to be able to check its real-time properties. However, it must be ensured that no relevant model detail is omitted and that the black-box abstraction is defined in a systematic and sound way.

An activity is defined by a triple of assumption, rely and guarantee. The assumption characterises the states when the activity may be operational. It is essentially an activity invariant satisfied at all times while the activity is running. The rely predicate states the operational conditions that must be satisfied by any execution steps of an environment that may happen during the execution of an activity. More specifically, it defines the maximum interference from the environment that the activity is ready to tolerate. Finally, the guarantee predicate defines an obligation that every execution step of an activity must fulfil.

We assume that the system state is completely determined by the values of its variables, \( v \). The system state space is denoted as \( \Sigma \). The system expressions are functions over the system variables interpreted over the system state, e.g., a system invariant \( I(v) \) is a function of the type \( \Sigma \rightarrow BOOL \). From now on we use the notions of the system state and its variables interchangeably.

**Definition 1 (Activity).** An activity is a tuple \((A, R, G)\), where \( A(v) \) is the assumption predicate, \( A : \Sigma \rightarrow BOOL \); \( R(v, v') \) and \( G(v, v') \) are the rely and guarantee predicates defined over the current state \( v \) and the next state \( v' \), \( R : \Sigma \times \Sigma \rightarrow BOOL \), \( G : \Sigma \times \Sigma \rightarrow BOOL \). An activity definition must satisfy the following conditions:

1. the assumption must define a non-empty set of operational states: \( \exists v \cdot A(v) \);
2. the rely must characterise a feasible environment interference:
   \( \forall v \cdot A(v) \Rightarrow \exists v' \cdot R(v, v') \);
3. the guarantee must be implementable: \( \forall v \cdot A(v) \Rightarrow \exists v' \cdot G(v, v') \);
4. the rely and guarantee must not violate the activity assumption:
   \( \forall v, v' \cdot A(v) \land (R(v, v') \lor G(v, v')) \Rightarrow A(v') \).

Let us denote the set of all system activities as \( A \). Now we ready to define a transition between two activities.
Definition 2 (Activity Transition). An activity transition is a tuple \((\text{src}, \text{dst}, \text{grd}, \text{act})\), where \(\text{src}\) is the source activity, \(\text{src} : A\), \(\text{dst}\) is the destination (target) activity, \(\text{dst} : A\); \(\text{grd}\) is the transition guard predicate, \(\text{grd} : \Sigma \rightarrow \text{BOOL}\), and \(\text{act}\) is the transition action defined as a next state relation, \(\text{act} : \Sigma \times \Sigma \rightarrow \text{BOOL}\). An activity transition \(t\) must satisfy the following conditions:

1. the transition guard must be compatible with the target activity assumption:
   \[ \forall v \cdot t.\text{grd}(v) \Rightarrow t.\text{src}.A(v) \]
2. the transition action must be feasible:
   \[ \forall v \cdot t.\text{grd}(v) \Rightarrow \exists v' \cdot t.\text{act}(v,v') \]
3. a transition must be compatible with the destination activity assumption:
   \[ \forall v,v' \cdot t.\text{grd}(v) \land t.\text{act}(v,v') \Rightarrow t.\text{dst}.A(v') \]

3.2 Definition of a Process

A process is assembled from a number of activities connected by activity transitions. This is understood as follows: a process behaves as a given activity until a transition happens and then, instantaneously, it continues as another activity. There is no concurrent behaviour within the process as it engages activities one at a time. Informally, instantaneous activity transition may be understood as a transition being a simple action that has no duration. This is, perhaps, the most important distinction between the activity and activity transition concepts.

Let \(A\) be a set of all system activities, and \(T\) be a set of all activity transitions. Then we can define a process in the following way:

Definition 3 (Process). A process is a tuple \((\text{Inv}, \text{Act}, \text{Trn}, \text{Rel}, \text{Grt}, \top_p, \text{Init})\), where \(\text{Inv}\) is the process invariant; \(\text{Act}\) and \(\text{Trn}\) are the sets of process activities and transitions respectively, \(\text{Act} \subseteq A\), \(\text{Trn} \subseteq T\); \(\text{Rel}\) and \(\text{Grt}\) are the process rely and guarantee conditions, \(\text{Rel} : \Sigma \rightarrow \text{BOOL}\), \(\text{Grt} : \Sigma \rightarrow \text{BOOL}\); \(\top_p\) is the initial process activity, \(\top_p \in \text{Act}\); and \(\text{Init}\) is the initialisation transition, \(\text{Init} \in \text{Trn}\). A process definition must satisfy the following conditions:

1. the initial process activity does not have any constraints:
   \[ \top_p.A = \text{true} \land \top_p.R = \text{true} \land \top_p.G = \text{false} \]
2. the initialisation transition always enabled: \(\text{Init}.\text{grd} = \text{true}\);
3. the process invariant must be implied by the assumption of the activities:
   \[ \forall a \in \text{Act} \cdot a.A(v) \Rightarrow \text{Inv}(v) \]
4. all the activities of a process must be reachable from the initial activity. Let \(T\) be the relation between activities defined by the activity transitions:
   \[ T = \{ t.\text{src} \mapsto t.\text{dst} \mid t \in \text{Trn} \} \]. Then, for any \(a \in \text{Act}\), \(\top_p \mapsto a \in T^*\), where \(T^*\) is a transitive closure of \(T\);
5. the process rely/guarantee attributes depend on the ones of process activities:
   (a) the process rely must be strong enough to satisfy the activity rely condition:
       \[ \forall a, v \cdot a \in \text{Act} \land \text{Inv}(v) \land \text{Rel}(v) \Rightarrow a.R(v) \]
   (b) the process guarantee must not be stronger than the activity guarantee:
       \[ \forall a, v \cdot a \in \text{Act} \land \text{Inv}(v) \land a.G(v) \Rightarrow \text{Grt}(v) \]
   (c) the process rely must be feasible:
       \[ \forall v \cdot \text{Inv}(v) \Rightarrow \exists v' \cdot \text{Rel}(v,v') \].
Here the process variables $v$ are the variables of all the contained process activities. According to the conditions (1) and (2), the initial system activity is essentially a placeholder for an undefined process state before the initialisation transition happens. The condition (3) means that the process invariant can be directly derived from the activity assumptions. The condition (4) requires that there cannot be any isolated activities in the process.

Finally, the condition (5) describes the relationships between the process rely/guarantee conditions and the ones of the process activities. It would be impractical to try to relate all the activities of a process with all other activities of the rest of a system. An abstraction of activity properties ($Rel$ and $Grt$) allows us to check the process compatibility at the level of process rely and guarantee conditions.

A PV model is assembled from a number of concurrently evolving processes. Two processes synchronise by simultaneously firing their activity transitions. During the synchronisation, time is frozen until both processes complete their transitions. Time flow restarts when both processes are engaged into their respective new activities. Synchronisation transitions are selected non-deterministically by matching transition-attached tags called channels.

**Definition 4 (Process View).** A Process View model is defined by a tuple $(I, P, C, S)$ where $I$ is a system invariant, $P$ is a set of processes, $C$ is a set containing all the channels used in process synchronization, and a function $S : T \rightarrow C \times \{!, ?\}$ attributes a channel and synchronization type to each process transition. A predefined channel $\tau : C$ denotes the absence of synchronization on a transition. A PV model definition must satisfy the following conditions:

1. the set of channels must be non-empty: $C \neq \emptyset$;
2. the overall model invariant must be compatible with the invariants of the individual processes: $\forall p \in P \cdot I \Rightarrow p.Inv$;
3. processes must be compatible:
   $\forall m, n, v \cdot m \in P \land n \in P \land m \neq n \Rightarrow (m.Grt(v) \Rightarrow n.Rel(v))$;
4. synchronized transitions must be composable. For any two distinct processes $p_1, p_2 \in P$ and any transitions $t_1, t_2$ that may be synchronised, i.e., such that $t_1 \in p_1.Tran \land t_2 \in p_2.Tran \land S(t_1) = \{c, !\} \land S(t_2) = \{c, ?\} \land c \neq \tau$, the following must hold:
   (a) $\exists v \cdot I(v) \land t_1.grd(v) \land t_2.grd(v)$;
   (b) $\forall v \cdot I(v) \land t_1.grd(v) \land t_2.grd(v) \Rightarrow \exists v' \cdot t_1.act(v, v') \land t_2.act(v, v')$;
5. initialisation transitions must be composable:
   $\exists v' \cdot \forall v, t \cdot t \in T \land t.src = \top_p \Rightarrow t.act(v, v')$.

The condition (6) checks that two synchronized transitions may be fused into a single one: for all the pairs of synchronized transitions it must be established that the transition guards and actions are non-contradictory in order to permit the execution of the transitions in a single atomic step. The condition (7) does the same for process initialisation transitions.
Our intention is to build PV models to conduct the verification of real-time properties of Event B models. For this we are first going to show how to link a PV model and an Event B model. Then we demonstrate how to reduce a PV model to a timed automata [16] suitable as an input to the Uppaal model checker [8].

4 Linking Event-B and Process View

This section discusses a mapping between a PV model and an Event B machine and how such mapping can be verified by discharging theorems in the first-order logic. We start by representing an Event B model as a special kind of a labeled state transition system.

**Definition 5 (Event-B Model).** An Event-B Model is defined by a tuple $(c, s, X, v, I, S_{Init}, E)$, where $c$ and $s$ are the model constants and sets respectively; $v$ are the model variables; $X(c, s)$ is a collection of axioms constraining $c$ and $s$; $I(c, s, v)$ is the model invariant limiting the possible states of $v$; $S_{Init}(c, s, v')$ is an initialisation action computing some initial values for the model variables; and $E$ is a set of model events. Moreover, each event is defined as a tuple $(H, S)$, where $H(c, s, v)$ is the event guard and $S(c, s, v, v')$ is the before-after predicate defining a relation between the current and the next states. For brevity, we would refer to the event guard as $H(v)$, the before-after predicate as $S(v, v')$, and the initialisation action as $S_{Init}(v')$.

We are interested in a notion of consistency that, for a given Event B model, would demonstrate that the corresponding PV model is its valid abstraction and hence the Event-B model enjoys all the properties of the PV counterpart. The overall strategy is following: a PV activity is related to a group of Event B events, while a PV activity transition would be realised by just a single event. A pair of synchronised transitions are also related to a single Event B event.

First we define an intermediate construct called *Mapping Model* for linking elements of a PV model to the elements of an Event B model.

**Definition 6 (Mapping Model).** Mapping Model relates the elements of given PV model $N$ and Event B model $M$ and is defined by a tuple $(L, ma, mt)$, where $L$ is a predicate relating the states of PV and Event models, i.e. $L(v, w) : \Sigma \times BState \rightarrow BOOL$. The functions $ma$ and $mt$ map, respectively, the activities and transitions of $N$ into the events $E$ of $M$, i.e. $ma : A_N \rightarrow \mathcal{P}(E)$, $mt : T_N \rightarrow E$. Here $BState$ is the state space of $M$, and $v, w$ are the variables of $N$ and $M$, correspondingly. $\mathcal{P}(E)$ is a set of all non-empty subsets of $E$, while $A_N, T_N$ stand respectively for all the activities and transitions of the model $N$.

Now we formulate the model consistency conditions that should be verified as proof obligations.

**Definition 7 (Mapping Consistency Conditions).** A PV model $N$, an Event B model $M$ and their Mapping Model $MM$ are consistent provided the following conditions hold:
1. All the events of $M$ are used in the mapping:
\[ \bigcup \text{ran}(\text{MM}ma) \cup \text{ran}(\text{MM}mt) = M.E; \]

2. For every activity $a$ such that $a \in A_N$ and every event $e$ such that $e \in \text{MM}ma(a)$, the following conditions must be demonstrated:
   (a) The event may be enabled only when the condition expressed by the activity is satisfied: $\forall v, w \cdot M.I(w) \land \text{MM}L(v, w) \land e.H(w) \Rightarrow a.A(v)$;
   (b) It must be established that the event satisfies the activity guarantee:
   \[ \forall v, w, w' \cdot M.I(w) \land a.A(v) \land \text{MM}L(v, w) \land e.H(w) \land e.S(w, w') \Rightarrow \exists v' \cdot \text{MM}L(v', w') \land a.A(v') \land a.G(v, v'); \]

3. For every transition $t$ such that $t \in T_N$, internal or synchronised, it must be shown that every associated event $e$, such that $e = \text{MM}mt(t)$, is a valid implementation of the transition:
   (a) $\forall v, w \cdot M.I(w) \land a.A(v) \land \text{MM}L(v, w) \land e.H(w) \Rightarrow t.grd(v)$;
   (b) $\forall v, w, w' \cdot M.I(w) \land a.A(v) \land \text{MM}L(v, w) \land e.H(w) \land e.S(w, w') \Rightarrow \exists v' \cdot \text{MM}L(v', w') \land t.grd(v) \land t.act(v, v')$;

4. For a pair of synchronised transitions $t_1$ and $t_2$, such that $t_1 \in T_N, t_2 \in T_N$, it is required to show that the transitions are mapped into the same event sets: $\text{MM}mt(t_1) = \text{MM}mt(t_2)$;

5. The INITIALISATION event must be mapped into a synchronised transition formed by the process initialisation events:
\[ \forall t \in T_N \cdot t.src = T_p \Rightarrow \text{MM}mt(t) = \text{INITIALISATION}. \]

Now we are able to formulate a property central to our proposal.

**Theorem 1.** For a triple of Event-B, PV and Mapping models satisfying Definitions 5, 6 and 7, it holds that every Event-B state transition $w \mapsto w'$ has a corresponding PV state transition $v \mapsto v'$ such that $L(v, w) \land L(v', w')$.

**Proof.** Here we only sketch the proof. The condition (7.1) of Definition 7 requires that all the Event B model events are covered by the mapping. Let us consider different cases of the event mapping. For every event related to an activity, the condition (7.2.b) guarantees that any state transition $w \mapsto w'$ in the Event B model corresponds to some state transition $v \mapsto v'$ in the Process View. Similarly, according to (7.3.b), an event realising an activity transition, internal or synchronised, defines a stronger next-state relation than the corresponding PV transition. Finally, initialisation event is treated as a special case of an event mapped into a synchronised transition (the condition (7.5)).

From Theorem 1 it follows that if we run a process view model and Event-B model side-by-side we are not going to observe deadlock that were already present in either model in isolation. This property gives us an ability to define the meaning of an Event-B augmented by a process view model.

**Definition 8 (Event-B/Process View Composition).** For some Event-B model $M$ and PV model $N$ satisfying Definitions 5, 6 and 7 by composition $M \parallel N$ we understand Event B model $M^c$ such that an event of event $M^c$ may fire only when its guard holds and the corresponding activity of $N$ is currently enabled.
In other words, a process view model is used to drive an Event B model and, thus, in a composed model we do not observe the traces disallowed by the process view model.

5 Example

The case study is concerned with a small part of the BePi system. Previously, a detailed Event-B development of the system has been constructed within the scope of the DEPLOY Project [15]. For the purpose of studying the applicability of the process view technique we limit ourselve to an abstraction of the system high-lightening the messaging handling rules of the Core Software Unit. The focus of the case study is on the specification of the reception of command messages from the ground control and the production and delivery of telemetry information.

The main parts of the system are TC pool, receiving and keeping incoming messages (called telecommands or TC); Decoder fetching messaging from the TC pool, running some validation checks and routing decoded messages to other components of the system; outgoing messages, called telemetry (TM), are kept in TM pool from where they are scheduled for the actual transmission to the ground control. There are several main classes of telemetry produced by the system: the decoder is obliged to acknowledge every incoming TC whether it is successfully decoded and handled or not. Every decoded message is passed on to an instrument which provides some custom report TM upon the execution of the request contained in TC. Some instruments may be switched into a mode where they generate telemetry on a periodic basis corresponding to the change in measurements obtained by an instrument. At the model level, it is convenient to think about this operational mode as a separate component of the system, called Health Monitor on the Figure 1. On the Figure 1, vertical arrows denote Event B refinement; the horizontal is the Event B - PV link; dashed lines indicate informal links and cycles the development. The overall goal of the case study
is to determine the relationship between the relative performance of the system components and the maximum time it takes to produce and a telecommand. One reason for conducting such analysis is to obtain an assurance the system is able to handling a certain a number of message per unit without having drop new or old message.

5.1 Event-B Model

The Event B development comprises a chain of refinement steps starting from a trivial statement and ending with rather detailed model that is still refined further although we leave is out of scope of the paper. We briefly discusses the few initial refinement steps relevant to the construction of a process view model.

Abstract model merely states that the globally observed behavior of the system is the production of telemetry data (TM) and its subsequent transmission to the ground control.

\[
\text{invariant } tmout \subseteq TM \\
\text{report } = \text{any } tm \text{ where } tm \notin tmout \text{ then } tmout := tmout \cup \{tm\} \text{ end} \\
\text{transmit } = \text{any } tm \text{ where } tm \in tmout \text{ then } tmout := tmout \setminus \{tm\} \text{ end}
\]

First refinement introduces the notion of telecommand (TC) and distinguishes between reactive reports and periodic TM production.

\[
\text{invariant } tcin \subseteq TC \land hk \in BOOL \\
\text{receive } = \text{any } tc \text{ where } tc \in tcin \text{ then } tcin := tcin \cup \{tc\} \text{ end} \\
\text{hk ref report } = \text{any } tm \text{ where } tm \notin tmout \land hk = \text{TRUE then } tmout := tmout \cup \{tm\} \text{ end}
\]

Second refinement splits TC and TM messages into several classes and shows how some of them are handled. For instance, certain kind of TC would trigger switching on or off of the periodic data production.

Third and fourth refinement steps model TC and TM pools as queues of messages so that the oldest message is one to be processed first.

Fifth refinement introduces further details on the workings of message pools and puts size constraints on TC and TM queues.

5.2 Process View Model

The next step in the case study is the construction of a process view model corresponding to the fifth refinement step of the Event B development. As a general guidance for structuring the system into processes we use the architectural diagram presented on Figure 1. Of course, there is more than one way to define a process view for a given Event B model. In our case, we also keep in mind the kind of properties we aim to verify.

From the system requirements it is known that Health Monitor, Decoder and Instruments are going to be realised as concurrent activities. As they compete
for the same resource (TM pool) there is no easy opportunity to merge these processes. It does not make sense to consider TC and TM pools as part of the same process as we know from the requirements that message reception and transmission hardware operate concurrently and at differing speeds.

**Platform sender** The first process of the PV model is concerned with the arrival of a messages from the spacecraft platform. On Figure 1 it corresponds to the left-most tc arrow. In the Event B model we did not distinguish between the construction of a new TC and saving a TC in the TC pool. In the process view model there are two distinct processes for these activities. The following an event saving some fresh TC into TC pool (fifth refinement):

```plaintext
receive = any tc where
   tc \notin \text{ran}(tcq) \land tc_{tl} - tc_{hd} < \text{TCQSIZE}
then
   tcq := tcq \cup \{ tc_{tl} + 1 \mapsto tc \} \& tc_{tl} := tc_{tl} + 1
end
```

where `tcq` is a queue variable and `tc_{tl}, tc_{hd}` are the queue head and tail markers. The event updates the contents of TC pool and the **Sender** process does not. Hence, the transition from `idle` to `done` must be synchronised with a transition in the TC pool process saving a new TC. The sender process is responsible for the generation of a fresh TC and thus it is the one initiating the synchronisation. The following is the definition of the process **Sender**.

```plaintext
idle = to done when new_tc' = newid sync newtc!
```

```plaintext
done = to idle
```

The model above describes a process with two activities: `idle` and `done`. Activity `idle` contains a single transitions switching to activity `done` synchronised on channel `newtc`. The transition signifies the arrival of a new TC. The TC value itself is stored in variable `new_tc`. Variable `new_tc` \in MSGID holds a new TC value computed by function `newid`. The sole purpose of the variable is to communicate the new TC value to the TC pool process.

**TC Pool** receives a new TC from **Sender** and saves it and in the internal buffer. The process also communicates with the **Decoder** process (Figure 1). It is necessary to explicitly demonstrate some essential properties of a message pool: a limited amount of memory for storing messages and the FIFO processing logic. Thus we have a rather detailed description of how message queue of the pool is realised. One difficulty here is that we are unable to simple copy the Event B definition of queue `tcq` since `tc_{tl}` and `tc_{hd}` cannot be restricted to finite domains. A slightly different design of a queue is used where read and write pointers are wrapped around the queue size. The process is mapped into two events of the Event B model: event `decode` reading a TC from the pool and event `receive` adding a new TC into the pool. As mentioned above, TC production is a transition synchronised with a transition of the **Sender** process and new TC is communicated through variable `new_tc`. To communicate a TC read from the pool we define a new channel `readtc`. Here we envisage that the **Decoder** process request a new TC from TC pool and thus the TC pool is the waiting party of the channel. Below is an excerpt from the process definition.
empty = to non_empty action
tc_buffer' = tc_buffer ⇥ {next_tc_inx ↦ new_tc}
next_tc_inx' = (next_tc_inx + 1) mod BUFF_SIZE
sync newtc?

full = to non_empty action
tc_buffer' = tc_buffer ⇥ {first_tc_inx ↦ EMPTY}
first_tc_inx' = (first_tc_inx + 1) mod BUFF_SIZE
sync readtc?

A definition of the process in the Uppaal is given on Figure 2. To define the mapping into the Event B we have to convert between two styles of queues. This forms a part of the Event B linking relation \[ L: \text{BUFF.SIZE} = \text{TCQ.SIZE} \land \text{ran(tc_buffer)} = \text{ran(tcq)} \land \text{next_tc_inx} = \text{tc_tl mod TCQ.SIZE} \land \text{first_tc_inx} = \text{tc_hd mod TCQ.SIZE} \land \ldots. \]

Decoder process fetches a message from TC pool and starts its decoding. If the decoding is successful it will pass on the TC to the Instrument process for the execution of the request contained in the decoded TC. The process interacts with several other processes: it gets TC from TC pool, delegates TC request execution to an instrument and adds a new TM in TM pool. An essential property of the process is the link between a TC consumed and a TM produced. This link allows us to associate the appearance of some TM message with a previously received TC message.

wait = to validate action curr_tc = tc_buffer(first_tc_inx) sync readtc!
validate = to acknowledge
to forward
forward = to execute sync req!
execute = to acknowledge sync res?
acknowledge = to wait action new_tm = tmid(curr_tc) sync newtm!

In the above, \( curr\_tc \) is a local process variable holding the value of TC being decoded. The process communicates new TM to the TM pool via variable \( new\_tm \). Constant function \( \text{tmid} \) derives a value of new TM from current TC. The process is associated with six events of the Event-B model. Since it is primarily concerned with the ordering of different stages of TC handling, the linking relation here is quite simple.

Health Monitor roughly corresponds to the behaviour captured by the \( hk \) event of the first refinement step. This process is a not part of the TC processing chain but rather autonomously generates new TM messages.

idle = to busy action new_tm' = newid2 sync newtm!

Due to space limitation we omit the discussion of TM Pool and Platform Receiver processes. These are similar to TC Pool and Platform Sender.

So far we have only glanced over the definition of the processes in the process view model. One essential part of the model is the set rely/guarantee conditions on the variables shared by the processes. There are several such variables in the model. Variable \( new\_tm \) (updated by Decoder and Health Monitor, read by TM pool) is used to communicate a fresh TM message. The guarantee of the updating processes states that the variable is unchanged or is set to a fresh
message value: \( \text{new\_tm}' = \text{new\_tm} \lor \text{new\_tm}' \notin \text{ran}(\text{tc\_buffer}) \). In its turn, \( \text{TM pool} \) relies that on the fact \( \text{new\_tm} \) is either the message at the tail of the queue or is a fresh message: \( \text{notempty} \implies \text{new\_tm}' \notin \text{ran}(\text{tc\_buffer}) \lor \text{new\_tm}' = \text{tc\_buffer}(\text{next\_tc\_inx} + \text{BUFF\_SIZE} - 1 \mod \text{BUFF\_SIZE}) \). Here \( \text{notempty} \) is the non-empty queue condition. Similar relies and guarantees are formulated for the writers and readers of \( \text{new\_tc} \). Two further shared variables are \( \text{tc\_buffer} \) and \( \text{first\_tc\_inx} \). These are updated only by the \( \text{TC pool} \) process. The \( \text{Decoder} \) process reads these variables and relies on the FIFO behaviour of the queue. This is expressed in the following set of guarantee conditions: when the buffer is not empty, \( \text{tc\_buffer}(\text{first\_tc\_inx}) \) points to a valid message; a valid (not EMPTY) message in \( \text{tc\_buffer} \) is never replaced by another valid message; EMPTY is never written over EMPTY.

With the relies and guarantees in place we are able to apply the verification conditions from Definitions 3 and 4. As far as it was possible we have tried to manually review all the conditions and convince ourselves of the validity of the model. It is certainly an extremely tedious and error prone exercise. In the small case study there were nearly fifty consistency and linking theorems. We are working on a tool that would allow us to enter process view models alongside Event B developments and automatically obtain all the relevant proof obligations.

5.3 Verification in Uppaal

The final step of the case study is the translation of the process view model into the Uppaal timed automata notation and the addition of clock variables. This is a fairly straightforward process as process view minus rely/guarantee predicates is essentially a finite automaton.

We only briefly summarise the verification results. A representation of a part of the process view model in the visual Uppaal notation is given on Figure 2. We were able to check that the reception of TC invariably leads to the production
of a new TM. We also run experiments to determine how various system parameters affect the key property of the system: the handling time of a TC message. One interesting case is the analysis of the interference from the Health Monitor process. We have found that if Health Monitor is working too fast the system may becomes unresponsive. On the other hand, after certain point, there is no advantage in slowing down Health Monitor (see Figure 3).

6 Discussion

Since real-time requirements have a direct impact on system dependability, verification of real-time properties and in particular, the integration of reasoning about functional and non-functional properties, have attracted significant research efforts. Time is an evident modelling challenge, that explains a substantial amount of work in the area of combining state-based methods, such as Action Systems and the B Method, and time modelling formalisms.

In [6], the concept of time is embedded into the B notation and time progress is modelled by equipping a machine with a clock and assuming that an event execution is not instantaneous but has a certain duration. The time duration for a specific event execution would be selected non-deterministically from within a region of values. Unfortunately, there are no readily available means for checking real-time properties of such models. In certain situations, timing properties may be successfully modelled with the set theoretic notation of the B Method and Event B [2–5]. The overall idea is to use one or more variables to representing clock readings and provide events to advance the clocks and thus simulate the time flow. The main modelling technique is expressing timing constraints as deadlines by adding timing guards to some critical events. An interesting (and, perhaps, worrying) consequence is that time is put under the control of a model: time is not allowed to progress past a deadline until an event scheduled for the deadline takes place. Thus, real-time properties are specified (postulated) rather than inferred from a system behaviour.
In this paper we proposed a practical approach to integrating verification of real-time properties into Event B modelling. Its development was driven by a pursuit of scalability and simplicity. As a result, we have developed a technique for building a process-based abstraction of an Event B model and employing such an abstraction in the verification of real-time properties. A set of well-formedness conditions ensures that the abstraction is built in a sound way. To achieve a semantic anchoring between process view and Event B models, we have put a special emphasis on expressing verification conditions as theorems in the Rodin Toolset. As a future work we are planning to restrict the axiomatisation the Process View definition to the smallest possible kernel from which the rest of the conditions could be deduced. We also planning to experiment with deriving real-time concurrent system implementations by refinement and distilling the guidelines on the constructing and using Process View model.

References

12. Rigorous Open Development Environment for Complex Systems (RODIN): (IST FP6 STREP project, online at http://rodin.cs.ncl.ac.uk/)
15. (EU-project DEPLOY) online at http://www.deploy-project.eu/.