Augmenting Event-B Modelling with Real-Time Verification

Alexei Iliasov, Alexander Romanovsky
Newcastle University
Newcastle Upon Tyne, UK
{*alexei.iliasov, alexander.romanovsky*}*@ncl.ac.uk*

Linas Laibinis, Elena Troubitsyna
Åbo Akademi University
Turku, Finland
{*linas.laibinis, elena.troubitsyna*}*@abo.fi*

Timo Latvala
Space Systems Finland
Espoo, Finland
Timo.Latvala@ssf.fi

***Abstract*—A large number of dependable embedded systems have stringent real-time requirements imposed on them. Analysis of their 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. We show how to extract a process-based view from an Event-B model that together with introducing time constraints allows us to obtain a timed automata model – an input model of Uppaal. We illustrate the approach by development and verification of the data processing software of the BepiColombo Mission.**

I. Introduction

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

Event-B modelling is focused on functional requirements. However, in design of embedded systems, non-functional requirements, such as real-time, play equally important role. Usually, real-time systems properties are evaluated at late development stages. This might incur costly redevelopment, if the real-time constraints are not met. Hence, it would be desirable to evaluate these properties as early as possible.

Real-time properties are to a great extent defined by the system concurrency model. Such a model is derived from the targeted system architecture. In the industrial setting, the system architecture is usually dictated either by the need to reuse existing components or by the constraints imposed by the customer. To facilitate construction of the targeted concurrency model, we propose to design an auxiliary model – a Process View (PV) – from an Event-B system model. While suppressing details of the functional behaviour, a PV model provides the explicit notions of processes and their synchronisation. We also define the proof obligations that guarantee that a PV model is a valid projection of the corresponding Event-B model. Then we augment the PV model with clocks and timing constraints and arrive at a timed automata model. The Uppaal model checker [2] is then used to verify liveness and real-time system properties.

Our approach is illustrated by an industrial case study – development of the data processing unit of the BepiColombo satellite undertaken within the EU FP7 project Deploy [3]. The initial development was undertaken by the company Space Systems Finland. The achieved results has allowed us to evaluate the impact of component performance and the frequency of data collection on system responsiveness.

Our approach aims at facilitating investigation of the real-time behaviour at the modelling stage rather than replacing simulation techniques for analysing real-time system characteristics at the implementation level. Thus it helps the designers to explore the impact of various architectural alternatives on real-time system properties.

The paper is organised as follows. Section II gives a very short overview of the Event-B formalism as well as presents Event-B development of the BepiColombo data processing unit. Section III describes a theoretical basis for constructing an explicit concurrency model – Process View (PV). In Section IV we briefly discuss application of the Uppaal model checker to verification of timed PV models. Finally, Section V concludes the paper with some final remarks.

II. Modelling in Event-B

A. Introduction to Event-B

The B Method [4] is a formal approach for development of highly dependable software. Event-B [5] is a formal framework derived from the B Method to model reactive systems. The Rodin platform [6] provides automated tool support for modelling and verification in Event-B.

In Event-B, a system model is an *abstract state machine* [1] defined as follows:

Definition *1 (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 (types) respectively;* $X(c, s)$ *is a collection of model axioms;* v *are the model variables;* $I(c, s, v)$ *is the model invariant limiting the possible states of* v*;* $S_{Init}(c, s, v')$ *is an initialisation action 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 a before-after predicate defining a relation between the current and next states.*

A general syntactic representation of Event-B models is given in Figure 1. While defining events, we adopt the

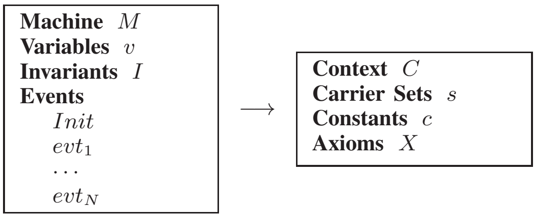

Figure 1. Event-B machine and context

following syntax:

$$evt \mathrel{\widehat{=}} \textbf{any } u \textbf{ where } H(v,u) \textbf{ then } act \textbf{ end},$$

where u are local event variables, the guard H is a state predicate, and the action act is an (either deterministic or nondeterministic) assignment to the state variables. A deterministic assignment has the standard syntax: $x := E(x, y)$. A nondeterministic assignment is denoted either as $x :\in Set$, where Set is a value set, or $x :| P(x, y, x')$, where P relates initial values of x, y to some final value of x'. As a result, x can get any value belonging to Set or according to P.

The occurrence of events represents the observable behaviour of the system. The guard defines the conditions under which the action can be executed, i.e., when the event is *enabled*. If several events are enabled at the same time, any of them can be chosen for execution nondeterministically.

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.

The semantics of an Event-B model is formulated as a number of *proof obligations*. For instance, the invariant proof obligations guarantee that the model invariant is preserved by all the events. The full list of proof obligations can be found in [1]. The Rodin platform [6] significantly facilitates verification in Event-B by automatically generating all proof obligations and providing both automatic and interactive provers to discharge them. Usually around 90% of proof obligations are discharged automatically.

B. Modelling and Refinement of the BepiColombo DPU

Space Systems Finland (SSF) is one of software providers for the European Space Agency mission BepiColombo [7]. The main mission goal is to carry various scientific studies to explore Mercury. SSF is responsible for developing software for an important part of the Mercury Planetary Orbiter – the data processing unit (DPU).

DPU consists of the core software (CS) and the software of two scientific instruments. CS communicates with the BepiColombo spacecraft to receive telecommands (TCs) from the spacecraft and transmit science and housekeeping telemetry data (TMs) back to it.

CS stores the received TCs in the TC buffer. CS is also responsible for validation of syntactical and semantical integrity of each TC. If validation fails, the corresponding TM is generated. A single TC might request to change the operational mode of a component, activate or deactivate scientific data generation, produce a housekeeping report etc. CS decodes TCs one-by-one and forwards the decoded TC to a required instrument. In its response, the instrument might perform a certain action and return an acknowledging TM. All the outgoing TMs are stored in the TM buffer.

This paper focuses on verification of real-time aspects of DPU based on its Event-B model. We aim at investigating the relationship between performance of the system components and the maximum time it takes to produce a TM. Hence, we will present only the bits of our Event-B development relevant to handling TCs and producing TMs.

Abstract model. In our initial specification we define the variable $tmout$, $tmout \subseteq TM$, to abstractly represent the TM buffer. The model has two events shown below. The event $report$ adds an element to $tmout$ modelling production of a new TM, while the event $transmit$ models transmission of a TM by removing it from the buffer.

```
report =
  any tm where tm ∉ tmout then tmout := tmout ∪ {tm} end
transmit =
  any tm where tm ∈ tmout then tmout := tmout \ {tm} end
```

First refinement. Our first refinement step introduces two new variables $tcin$ and $hkstatus$. The variable $tcin$, $tcin \subseteq TC$, models the buffer of incoming TCs, while $hkstatus$, $hkstatus \in BOOL$, shows whether periodic production of housekeeping TMs is activated. We introduce a new event $receive$ to model receiving a TC. The event $tchandling$ refines the event $report$ in order to update the housekeeping status. The event hk is also a refinement of $report$. It models production of housekeeping TMs.

```
receive =
  any tc where  tc ∉ tcin then  tcin := tcin ∪ {tc} end
tchandling  =  any tm, tc where
                  tc ∈ tcin ∧ tm ∉ tmout
               then
                  tmout := tmout ∪ {tm} ||
                  tcin := tcin \ {tc} || hk :∈ BOOL
               end
hk          =  any tm where
                  tm ∉ tmout ∧ hk = TRUE
               then
                  tmout := tmout ∪ {tm}
               end
```

Second refinement. At this refinement step we further elaborate on TC handling and TM production. We introduce a new variable, $curr\_tc$, to contain the current TC handled by DPU. The new boolean variable $decoded$ reflects whether the last validation of $curr\_tc$ has been successful. The new event $decode$ abstractly models choice of the TC to be handled and its validation. If TC validation fails, the new event $report\_error$ becomes enabled. It generates an outgoing TM with its value from the set

TM_DECODE_ERROR. Otherwise, the validated TC may, e.g., (de)activate generation of housekeeping data, as modelled in the new event $compute\_hk\_flag$.

```
decode =
  any tc where
    tc ∈ tcin ∧ curr_tc = NO_TC
  then
    decoded :∈ BOOL ∥ curr_tc := tc
  end
report_error =
  any tm where
    decoded = FALSE ∧ curr_tc ≠ NO_TC ∧ report = NO_TM
    ∧ tm ∈ TM_DECODE_ERROR \ tmout
  then
    report := tm
  end
compute_hk_flag =
  any hfun where
    report ≠ NO_TM ∧ curr_tc ∈ {TC_HK_ON, TC_HK_OFF}
    hfun = {TC_HK_ON ↦ TRUE, TC_HK_OFF ↦ FALSE}
  then
    hk_flag := hfun(curr_tc)
  end
```

Third and fourth refinement steps. In our abstract model the TC and TM buffers are modelled as sets. Next we refine this abstraction by replacing the sets by the corresponding queues. At the third refinement step we perform the data refinement replacing the TC buffer, the set variable $tcin$, by the queue $tcqueue$. In the resulting model, the following gluing invariant is used to prove refinement correctness:

$$tcqueue \in tc\_hd + 1..tc\_tl \rightarrowtail TC \;\;\wedge\;\; tc\_hd \in \mathbb{N} \;\;\wedge$$
$$tc\_tl \in \mathbb{N} \;\;\wedge tcin = ran(tcqueue) \;\;\wedge\;\; tc\_hd \leq tc\_tl \;\;\wedge$$
$$curr\_tc \neq NO\_TC \Rightarrow tc\_hd < tc\_tl \;\;\wedge$$
$$curr\_tc \neq NO\_TC \Rightarrow curr\_tc = tcqueue(tc\_hd + 1)$$

In the events we replace operations over the set $tcin$ by the corresponding operations over $tcqueue$. This allows us to ensure that TCs are handled in the ”first-in-first-out” order.

In the fourth refinement step we perform a similar data refinement of the TM buffer by replacing $tmout$ with $tmqueue$. Moreover, we introduce a new event $execute$. This event abstractly models the computations required to produce a housekeeping TM. Two new boolean variables $execution$ and $executed$ reflect whether the computations have been respectively requested and performed.

```
execute =
  when
    executed = FALSE ∧ curr_tc ≠ NO_TC ∧
    decoded = TRUE ∧
    curr_tc ∉ {TC_HK_ON, TC_HK_OFF}
  then
    executed :∈ BOOL ∥ execution := TRUE
  end
```

Fifth refinement. In the last refinement step we present here, we restrict sizes of the TC and TM queues. To achieve that, we introduce the constants TC_QUEUE_SIZE and TM_QUEUE_SIZE and define the constraining invariants as follows:

$$tc\_tl - tc\_hd \leq TC\_QUEUE\_SIZE \;\;\wedge$$
$$tm\_tl - tm\_hd \leq TM\_QUEUE\_SIZE$$

Due to the space limit we give only an outline of model events. The complete model can be found at [8].

MACHINE m5	
EVENTS	
$decode$	choice of a next TC and its validation
$report\_error$	a TM about failed TC validation
$report\_success$	a TM about successful TC validation
$compute\_hk\_flag$	switching on/off housekeeping TMs
$execute$	TM generation process
$tchandling$	handling a TC from the TC buffer
hk	generating a housekeeping TM
$transmit$	sending a TM (from the TM buffer)
$receive$	receiving a new TC (into the TC buffer)

The obtained Event-B model of DPU is still very abstract. In [9] the model was refined further to introduce realistic mechanisms for TC validation, TC decoding, and TM generation. However, in this paper we omit a discussion of the entire development and use the model $m5$ as a basis for analysing the real-time characteristics of DPU. Next we will briefly outline the principles of constructing an explicit concurrency model and demonstrate how to create such a model for the obtained specification of DPU.

III. Constructing an Explicit Concurrency Model

As discussed in Section I, the concurrency model that the system implements has a crucial impact on its real-time properties. To construct an explicit concurrency model, we propose a rely-guarantee [10] based framework called *Process View*. Next we overview the basic definitions and propose the proof obligations that guarantee consistency between corresponding Process View and Event-B models.

Process View (PV) is a specific projection of an Event-B model with explicit concurrency and synchronisation. An intermediate PV model covers much of the semantic gap between an event-based system characterisation, provided by Event-B, and timed automata – our formalism of choice for conducting timed analysis. The reason we propose a new approach rather than adopt one of the existing notations is to ensure the practicality and scalability of the verification routine. The PV design and its structuring primitives are dictated by the Event-B proof semantics. An important consideration is also the use of the Event-B infrastructure, in particular its automated proving, to deal with the verification conditions introduced by the construction of a PV model.

Formally, a PV model is a separate modelling artefact. It is linked with an Event-B model by a number of verification conditions inducing a simulation relation between the models. An Event-B specification is said to simulate a corresponding PV model. In other words, a PV model is some abstraction of an Event-B specification. We argue that such an abstraction of the Event-B model behaviour allows us to reason about its timed properties.

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

A. Process View

The simplest form of a process is called an *activity* – an abstract characterisation of a piece of functionality. 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. The rely states the operational conditions that must be satisfied by any changes in an environment during execution of an activity, i.e., the maximum interference from the environment that the activity can tolerate. Finally, the guarantee defines an obligation that every execution step of an activity must fulfil.

Definition 2 *(Activity): Let Σ be the system state space. 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', $R, G : \Sigma \times \Sigma \rightarrow BOOL$.*

An activity must also satisfy a number of conditions (omitted here for brevity) that ensure that the set of operational states are not empty and assumption, rely and guarantee are not contradictory with each other.

Let $\mathcal{A}$ be the set of all system activities. Then we can define a transition between two activities as follows:

Definition 3 *(Activity Transition): An activity transition is a tuple (src, dst, grd, act), where src is the source activity, $src : \mathcal{A}$, dst is the destination (target) activity, $dst : \mathcal{A}$; grd is the transition guard predicate, $grd : \Sigma \rightarrow BOOL$, and act is the transition action defined as a next state relation, $act : \Sigma \times \Sigma \rightarrow BOOL$.*

While defining a transition, we should also ensure that the transition guard is compatible with the target activity assumption, the transition action is feasible, and the transition is compatible with the destination activity assumption.

Activities connected by activity transitions form a *process*. There is no concurrent behaviour within the process as it engages into activities one at a time. Let $\mathcal{A}$ be a set of all system activities, and $\mathcal{T}$ be a set of all activity transitions. Then we can define a process in the following way:

Definition 4 *(Process): A process is a tuple $(Inv, Act, Trn, Rel, Grt, \top_p, Init)$, where Inv is the process invariant; Act and Trn are the sets of process activities and transitions respectively, $Act \subseteq \mathcal{A}$, $Trn \subseteq \mathcal{T}$; Rel and Grt are the process rely and guarantee conditions, $Rel, Grt : \Sigma \rightarrow BOOL$; $\top_p$ is the initial process activity, $\top_p \in Act$; and $Init$ is the initialisation transition, $Init \in Trn$.*

There is a number constraints that the definition of a process should satisfy. For instance, we should verify that the initial process activity is not empty and all other activities are reachable from it. We should prove that the activity assumptions imply the process invariant. Moreover, we need verify the relationships between the rely/guarantee of process and its constituting activities. An abstraction of activity properties (Rel and Grt) allows us to check compatibility at the level of the process rely and guarantee conditions.

An explicit concurrency model that we aim at building – a PV model – is assembled from a number of concurrently running processes. Two processes synchronise by simultaneously firing their activity transitions. Synchronisation is achieved by matching transition tags called *channels*.

Definition 5 *(Process View): A Process View model is defined by a tuple (I, P, C, S), where I is the system invariant, P is a set of processes, C is a set containing all the synchronisation channels, and a function $S : \mathcal{T} \rightarrow C \times \{!, ?\}$ attributes a channel and the synchronisation type to each process transition. The predefined channel $\tau : C$ denotes the absence of synchronisation on a transition.*

While creating a PV model, we should verify that composed processes are compatible with each other. For instance, this includes checking that the guarantee of one process implies the rely of other process and the overall model invariant implies the invariants of the processes. We also should guarantee that two synchronised transitions may be fused into a single one: the transition guards and actions should be non-contradictory in order to permit the execution of the transitions in a single atomic step.

B. Consistency Between Process View and Event-B

To ensure consistency for a given Event-B model, we should demonstrate that the corresponding PV model is its valid abstraction. Intuitively, we should show that any PV activity is related to a group of Event-B events, while there is an Event-B event for each activity transition. Moreover, we should establish a correspondence between a pair of synchronised transitions and an Event-B event.

To derive proof obligations for establishing consistency between PV and Event-B, we first define an intermediate construct called *Mapping Model*. It links elements of a PV model to those of an Event-B model defined in Section II.

Definition 6 *(Mapping Model): Let N be a PV model and M be an Event-B model. Then the Mapping Model is defined by a tuple (L, ma, mt), where L relates the states of PV and Event models, $L : \Sigma \times BState \rightarrow BOOL$. Here $BState$ is the state space of M. The functions ma and mt map, respectively, the activities and transitions of N into the events of M, $ma : \mathcal{A}_N \rightarrow \mathbb{P}1(E)$, $mt : \mathcal{T}_N \rightarrow E$. Here $\mathcal{A}_N, \mathcal{T}_N$ stand respectively for all the activities and transitions of the model N.*

Now we can formulate the model consistency conditions:

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*:
   $\bigcup(ran(MM.ma)) \cup ran(MM.mt) = M.E$;

2) *for every activity a, such that* $a \in \mathcal{A}_N$*, and every event* e*, such that* $e \in \mathit{MM}.ma(a)$*, the following conditions must be demonstrated*:
   a) *the event may be enabled only when the activity assumption is satisfied*: $\forall v, w \cdot M.I(w) \wedge \mathit{MM}.L(v, w) \wedge e.H(w) \Rightarrow a.A(v)$;
   b) *it must be established that the event satisfies the activity guarantee*: $\forall v, w, w' \cdot M.I(w) \wedge a.A(v) \wedge \mathit{MM}.L(v, w) \wedge e.H(w) \wedge e.S(w, w') \Rightarrow \exists v' \cdot \mathit{MM}.L(v', w') \wedge a.A(v') \wedge a.G(v, v')$;
3) *for every transition* t*, such that* $t \in \mathcal{T}_N$*, it must be shown that every associated event* e*, such that* $e = \mathit{MM}.mt(t)$*, is a valid implementation of the transition:*
   a) $\forall v, w \cdot M.I(w) \wedge a.A(v) \wedge \mathit{MM}.L(v, w) \wedge e.H(w) \Rightarrow t.grd(v)$;
   b) $\forall v, w, w' \cdot M.I(w) \wedge a.A(v) \wedge \mathit{MM}.L(v, w) \wedge e.H(w) \wedge e.S(w, w') \Rightarrow \exists v' \cdot \mathit{MM}.L(v', w') \wedge t.grd(v) \wedge t.act(v, v')$;
4) *for a pair of synchronized transitions* t_1 *and* t_2*, such that* $t_1 \in \mathcal{T}_N$*,* $t_2 \in \mathcal{T}_N$*, it is required to show that the transitions are mapped into the same* event: $\mathit{MM}.mt(t1) = \mathit{MM}.mt(t2)$;
5) *The* INITIALISATION *event must be mapped into a synchronised transition of the process initial activities*: $\forall t \in \mathcal{T}_N \cdot t.src = \top_p \Rightarrow \mathit{MM}.mt(t) = \text{INITIALISATION}$.

Theorem 1 then formalises the consistency conditions between Event-B and PV models.

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

The theorem proof as well as the complete list of PV model formal conditions can be found in [8]. Let us now exemplify our approach by constructing a PV model of DPU.

C. A Process View Model of DPU

Our verification goal is to determine the ability of the system to handle a certain number of TCs per time unit. More specifically, we aim at estimating the correspondence between the rate of TC arrival, the speed of TC decoding, the rate of housekeeping data production and the capacity of the TC and TM buffers. To reason about a relative speed of the TC decoding subsystem, we define it as a process in a PV model. Similarly, for each mentioned subsystem, we define a separate PV process. In a PV model, processes operate at arbitrary speeds. Later, when a timed model is created, we will introduce the cost (time) of process activities.

Below we show construction of a part of the PV model concerning TC arrival. In the Event-B model $m5$, this corresponds to just one event *receive*:

receive = **any** tc **where**
  $tc \notin ran(tcq) \wedge tc\_tl - tc\_hd < \text{TCQSIZE}$
then
  $tcq := tcq \cup \{tc\_tl + 1 \mapsto tc\} \parallel tc\_tl := tc\_tl + 1$
end

We construct a PV abstraction that achieves the same effect as a synchronised transition of two PV processes. The first process deals solely with announcing arrival of a new TC and giving it a name, which is stored in the global variable new_tc. There is a finite set of possible TC names, *idset*, recycled among all the processes.

process *sender* =
  rely $new\_tc' = new\_tc \wedge idset \subseteq idset'$
  guarantee $idset' \subseteq idset \wedge new\_tc' \in (idset \setminus idset') \cup \{new\_tc\}$
  ...

The body of *sender* is essentially one transition generating a fresh TC. We introduce an additional helper transition and activity to play the role of a 'delay', i.e., simulate the time between appearance of two TCs:

idle = **assume** $idset \neq \varnothing$
  to *done* **action**
    $new\_tc' = \max(idset) \wedge idset' = idset \setminus \max(idset)$
  sync $newtc!$
done = **to** *idle* **when** $idset \neq \varnothing$

The assumption $idset \neq \varnothing$ states that the activity *idle* should never encounter a situation when there are no fresh message ids. A fresh new_tc is computed by taking a distinguished element (here, a maximum) from $idset$. The activity *done* may switch to the activity *idle* if there is a fresh name in $idset$. At the system level this means that the rest of the system is prepared for new TCs.

In the Event-B model, a fresh TC in saved in the buffer. In the PV model, we we explicitly define a process *tcpool* that manipulates this buffer. The process saves new TCs in the buffer (variable $tcbuffer$) and makes previously saved TCs available to other processes. It is a circular buffer with separate pointers for the first and the last elements.

process *tcpool* =
  rely $new\_tc' = new\_tc \vee new\_tc' \notin elts(tcbuffer)$
  guarantee $(tcbuffer' \setminus tcbuffer) \cap idset = \varnothing$
  ...

Here the set $elts(tcbuffer)$ denotes the contents of the buffer, while the variable $next\_tc$ stores the index of the last message. The process guarantee requires that the buffer does not keep fresh (non-existent) messages. The process rely states that the buffer takes new TCs from the variable new_tc, the value of which has to be a valid fresh TC.

The body of process *tcpool* is made of three activities: *empty*, *not_empty* and *full* corresponding to the respective buffer states. An important part of *tcpool* is the interaction with the *sender* process. For instance, when the buffer is empty, it may become non-empty by receiving a new TC:

empty = **assume** $next\_tc = first\_tc$
  to *non_empty* **action**
    $tcbuffer' = tcbuffer \Leftarrow \{next\_tc \mapsto new\_tc\}$
    $next\_tc' = (next\_tc + 1) \bmod \text{BUFF\_SIZE}$
  sync $newtc?$

Here *first_tc* is the index of the oldest message.

The transition action is 'glued' with the corresponding action of the sender process (via the synchronisation channels) and the overall effect is similar to that of the event *receive*. In fact, our PV model exhibits exactly the same behaviour as its Event-B part. The models differ only in the way they represent the buffer and treat fresh messages.

Due to the space constraints we do not discuss the details of other PV processes. They can be found in [8].

IV. Towards Real-Time Verification with Uppaal

A. From Process View to Timed Automata

In this paper we use a PV model as an intermediate step towards a timed model suitable for checking the desired real-time properties. Our approach is driven by the pursue of scalability and industrial relevance. Hence, to perform verification of real-time properties, we have to chose a framework that is scalable and well-maintained. Timed automata [11] and the verification tool Uppaal [2] satisfy these criteria.

Timed automata [11] is a formalism with an explicit model of time. It is based on a finite automaton characterising the system behaviour via a number of states (locations) and state transitions. An array of real-valued clocks is introduced for time keeping. All the clocks progress synchronously and can be independently reset. State transitions are instantaneous, thus time advances only while the system stays in a given state. A logical expression called a location invariant sets the boundaries for time progress. Time constraints can also appear is state transitions, in the form of a predicate on clock values and a list of clock resets.

We propose the following technique for augmenting a PV model with time. The PV model is extended with a vector of real-valued clocks C; each activity is extended with time invariant $\phi(C)$; each transition is extended with a tuple of time guard $\omega(C)$ and clock reset θ, $\theta \subseteq C$. For each activity such that its guarantee G permits state update (i.e., $\exists v, v' \cdot v' \neq v \wedge G(v, v')$), there added a self-transition with guard $\top$ and action G. The relies, guarantees and assumptions of processes and activities are removed. This is justified by two reasons. First, the relies and guarantees are merely verification assistants, they do not describe actual behaviour. Second, state evolution inside of an activity is completely covered by the addition of a self-transition. Activities are treated as named states and activity transitions as state transitions.

Next we present the results of augmenting the PV model of DPU with time and show how to verify the desired real-time properties using Uppaal.

B. Real-Time Verification of the DPU

A representation of a part of the PV model augmented with time in the visual Uppaal notation is given in Figure 2.

For verification of timing properties of a Uppaal model, a simplified version of CTL (Computation Tree Logic) is used. We are mostly interested in verifying liveness and time-bounded reachability properties. In particular, we need to verify that, for any received TC, the corresponding TM is eventually returned. In CTL, this can be expressed as

$$(\mathsf{new\_tc} == \mathsf{id}) \;\longrightarrow\; (\mathsf{last\_tm} == \mathsf{id})$$

where $\longrightarrow$ is the "leads-to" operator, and id is some TC id.

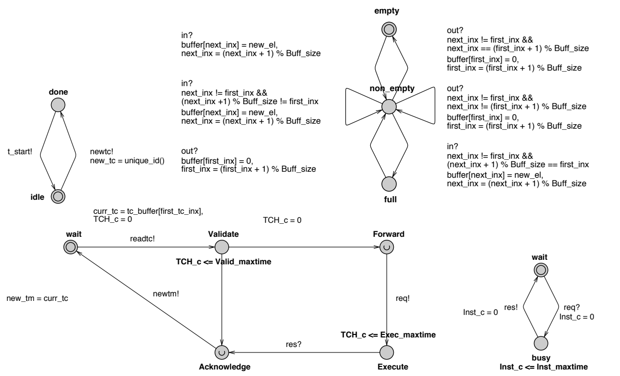

Figure 2. The BepiColombo process view model Uppaal notation.

Uppaal allows us to add various timing constraints and then check time-bounded reachability properties using the values of clock variables. One way to define such a property in our case is as follows:

$$\mathsf{A[]}\ (\mathsf{last\_tm} == \mathsf{id}\ \&\&\ \mathsf{Obs1.stop})\ \mathsf{imply}\ (\mathsf{Obs1\_c} < \mathsf{upper\_bound})$$

where $\mathsf{A[]}$ means "Always, for any execution path", while $\mathsf{Obs1}$ is a special process that starts the clock $\mathsf{Obs1\_c}$, whenever a TC command with id is received, and stops it, once the corresponding TM is returned. This property essentially verifies the maximal response time of the system.

The value of $\mathsf{upper\_bound}$ depends on concrete quantitative system parameters. In our case, such parameters are

- the size of buffers for storing TCs and TMs;
- the worst execution time (WET) for the instrument responding to the forwarded TC;
- the WET for validation of the arrived TC;
- the period of the process regularly generating housekeeping data returned as additional TMs;
- the maximal delivery delay for an outgoing TM, etc.

Naturally, different combination of these parameters may lead to quite different response times. In particular, we have noticed that the buffer size almost linearly correlates with the required time. This gave us the idea to use it as a time unit while verifying other parameter correlations.

Figure 3 illustrates how specific values of the period for production of housekeeping data affects the overall TC handling time. The time values are given as multiples of the buffer size. It is interesting to see that small values of the period (i.e., very frequent interference) makes the system essentially unresponsive, while bigger values form a "plateau" indicating that this interference becomes a constant.

We believe that our experiment with real-time analysis of DPU allowed us to identify a strategy for integrating real-time verification into the formal development process. We

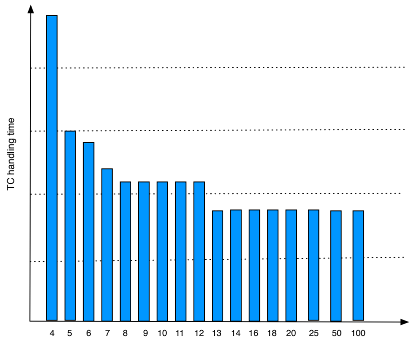

Figure 3. The relationship between the period of housekeeping TM production (X-axis) and worst TC handling time (Y-axis).

obtained the desired correlation between system responsiveness and various characteristics of its components. Such a correlation paves the path to optimising concurrency model and enables efficient design space exploration.

V. Related Work and Conclusions

Since real-time requirements have a direct impact on system dependability, verification of real-time properties has attracted significant research efforts. In particular, a substantial amount of work is done in the area of combining state-based methods and time modelling formalisms.

In [12], 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 has a certain (non-deterministically selected) duration. Unfortunately, there are no available means for checking real-time properties of such models. In certain situations, timing properties may be successfully modelled within the B Method and Event-B [13]–[16]. The overall idea is to use one or more variables to represent clock readings as well as provide events to advance the clocks. The main modelling technique is expressing timing constraints as deadlines by adding timing guards to some critical events. A worrying consequence is that time is put under the control of a model: time is not allowed to progress past a deadline until a scheduled event takes place. Thus, real-time properties are postulated rather than inferred from the 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.

Our approach has been validated in the context of the Deploy project [3]. Space Systems Finland together with the academic partners has conducted an exploratory study aimed at finding a scalable and useful approach to integrating real-time analysis into formal development. In this paper we have only described the main stages of this approach – from Event-B modelling via Process View to timed automata – and presented the semantic links between the stages.

In our approach we have put a special emphasis on defining a set of well-formedness conditions ensuring soundness of new abstractions. To achieve a semantic anchoring between PV and Event-B models, we have formally expressed verification conditions as theorems to be verified in the Rodin platform. As a future work, we are planning to experiment with deriving real-time concurrent system implementations by refinement and distilling the guidelines on the constructing and using PV models. There is also an ongoing work on developing a plug-in that integrates construction of a PV model and its verification in Rodin.

References

[1] C. Metayer, J. Abrial, and L. Voisin, Eds., *Rodin Deliverable D7: Event B language*. Project IST-511599, School of Computing Science, Newcastle University, 2005.

[2] J. Bengtsson, K. Larsen, F. Larsson, P. Pettersson, and W. Yi, "Uppaal - a tool suite for automatic verification of real-time systems," in *Proceedings of the DIMACS/SYCON Workshop on Hybrid systems*. Springer-Verlag, 1996, pp. 232–243.

[3] EU-project DEPLOY, online http://www.deploy-project.eu/.

[4] J. R. Abrial, *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 2005.

[5] J.-R. Abrial, *Modeling in Event-B*. Cambridge University Press, 2010.

[6] "Event-B and the Rodin Platform," http://www.event-b.org/.

[7] Factsheet: BepiColombo. ESA Media Center, Space Science, 15.01.2008, online at http://www.esa.int/esaSC/SEMNEM3MDAF_0_spk.html.

[8] A. Iliasov, L. Laibinis, E. Troubitsyna, A. Romanovsky, and T. Latvala, "Augmenting Event-B Modelling with Real-Time Verification," TUCS Technical Report, 2011, http://tucs.fi.

[9] OBSW formal development in Event-B, online at http://deploy-eprints.ecs.soton.ac.uk/view/type/rodin=5Farchive.html.

[10] C. B. Jones, "Specification and design of (parallel) programs," in *IFIP83*, 1983, pp. 321–332.

[11] R. Alur and D. Dill, "A theory of timed automata," *Theoretical Computer Science*, vol. 126(2), pp. 183–235, 1994.

[12] K. Lano, *The B Language and Method: A Guide to Practical Formal Development*. Springer-Verlag New York, Inc., 1996.

[13] 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)*, January 2002.

[14] J. Rehm, "A Method to Refine Time Constraints in Event-B Framework," in *Proceedings of AVoCS*, 2006.

[15] J. W. Bryans, J. S. Fitzgerald, A. Romanovsky, and A. Roth, "Patterns for Modelling Time and Consistency in Business Information Systems," in *Int. Conference on Engineering of Complex Computer Systems*. IEEE Computer Society, 2010.

[16] D. Cansell, D. Méry, and J. Rehm, "Time Constraint Patterns for Event B Development," in *Formal Specification and Development in B, 7th Int. Conference of B Users*, 2007.