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 roles. Usually, real-time systems properties are evaluated at late development stages. This might incur costly rework, 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


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 \((c, s, X, v, I, S_{\text{init}}, E)\), 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 \(v\); \(S_{\text{init}}(c, s, v')\) 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
where number of Event-B supports data refinement, allowing us to replace and variables into the abstract specification. Moreover, statements, each refinement step typically introduces new events and functional requirements. While capturing more detailed requirements, the event 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 telemetry data (TMs) from the spacecraft and transmit science and housekeeping telemetry data (TM) 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 ⊆ 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.

#### First refinement

Our first refinement step introduces two new variables `tecin` and `hkstatus`. The variable `tecin`, `tecin ⊆ TC`, models the buffer of incoming TCs, while `hkstatus`, `hkstatus ∈ BOOL`, shows whether periodic production of housekeeping TMs is activated. We introduce a new event `receive` to model receiving a TC. The event `techandling` 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.

```plaintext
receive = any tc where tc ∉ tm then tcin := tcin ∪ {tc} end

shandling = 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, `cur_tec`, to contain the current TC handled by DPU. The new boolean variable `decoded` reflects whether the last validation of `cur_tec` 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

---

**Figure 1. Event-B machine and context**

- **Machine** $M$
- **Variables** $v$
- **Invariants** $I$
- **Events** $\begin{align*}
  \text{Init} \\
  \text{evt}_1 \\
  \cdots \\
  \text{evt}_N
  \end{align*}$
- **Context** $C$
- **Carrier Sets** $s$
- **Constants** $c$
- **Axioms** $X$
Due to the space limit we give only an outline of model events. The complete model can be found at [8].

\[\text{MACHINE m5}\]

**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
- `transmit` generating a housekeeping TM
- `receive` sending a TM (from the TM 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 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 ab-
stract 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 ful-
fill.

Definition 2 (Activity): Let \( \Sigma \) be the system state space.
An activity is a tuple \((A, R, G)\), where \( A(v) \) is the assump-
tion predicate, \( A \) : \( \Sigma \rightarrow \{\text{BOOL}\} \); \( R(v, v') \) and \( G(v, v') \) are the
rely and guarantee predicates defined over the current
state \( v \) and the next state \( v' \), \( R, G : \Sigma \times \Sigma \rightarrow \{\text{BOOL}\} \).

An activity must also satisfy a number of conditions
(omitted here for brevity) that ensure that the set of op-
erational 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 \((\text{src}, \text{dst}, \text{grd}, \text{act})\), where \text{src} is the source activ-
ity, \text{dst} is the destination (target) activity, \text{dst} : \mathcal{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}\}.

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 tran-
sition 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 \((\text{Inv}, \text{Act},
\text{Trn}, \text{Rel}, \text{Grt}, \mathcal{T}_p, \text{Init})\), where \text{Inv} is the process invari-
ant; \text{Act} and \text{Trn} are the sets of process activities and
transitions respectively. \text{Act} \subseteq \mathcal{A}, \text{Trn} \subseteq \mathcal{T}; \text{Rel} and \text{Grt}
are the process rely and guarantee conditions, \text{Rel}, \text{Grt} : \Sigma
\rightarrow \{\text{BOOL}\}, \mathcal{T}_p \subseteq \text{Act}; \text{and Init} is the initialisa-
tion transition, \text{Init} \in \text{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 pro-
cess and its constituting activities. An abstraction of activity
properties (\text{Rel} and \text{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 simulta-
neously 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 invari-
ant, \( P \) is a set of processes, \( C \) is a set containing all the
synchronisation channels, and a function \( S : T \rightarrow C \times \{!, ?\} \)
attaches a channel and the synchronisation type to each
process transition. The predefined channel \( ! \) denotes
the absence of synchronisation on a transition.

While creating a PV model, we should verify that com-
piled processes are compatible with each other. For in-
stance, 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, \( L : \Sigma \times \text{BSate} \rightarrow \{\text{BOOL}\} \). Here
\text{BSate} is the state space of \( M \). The functions \text{ma} and
\text{mt} map, respectively, the activities and transitions of \( N
\) into the events of \( M \), \text{ma} : \mathcal{A}_N \rightarrow \mathcal{P}(1\{E\}), \text{mt} : \mathcal{T}_N \rightarrow \{E\}. \text{Here}
\text{AN}, \text{TN} 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
\text{MM} are consistent provided the following conditions hold:

1) all the events of \( M \) are used in the mapping:
\( \bigcup \{\text{ran}((\text{MM,ma})) \cup \text{ran}((\text{MM,mt})) = M\}.\)
2) for every activity \( a \), such that \( a \in A_N \), and every event \( e \), such that \( e \in MM.mna(a) \), the following conditions must be demonstrated:

a) the event may be enabled only when the activity assumption is satisfied: \( \forall v, w \cdot M.I(v) \land MM.L(v, w) \land e.H(v) \Rightarrow a.A(e) \);

b) it must be established that the event satisfies the activity guarantee:
\[
\forall v, w, w' \cdot M.I(v) \land a.A(e) \land MM.L(v, w) \land e.H(v) \land e.S(w, w') \Rightarrow \\
\exists v' \cdot MM.L(v', w') \land a.G(v, v');
\]

3) for every transition \( t \), such that \( t \in T_N \), it must be shown that every associated event \( e \), such that \( e = MM.mit(t) \), is a valid implementation of the transition:

a) \( \forall v, w \cdot M.I(v) \land a.A(e) \land MM.L(v, w) \land e.H(v) \Rightarrow t.grd(v); \)

b) \( \forall v, w, w' \cdot M.I(v) \land a.A(e) \land MM.L(v, w) \land e.H(v) \land e.S(w, w') \Rightarrow \\
\exists v' \cdot MM.L(v', w') \land t.grd(v) \land t.act(v, v'); \)

4) for a pair of synchronized 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:
\( MM.mt(t_1) = MM.mt(t_2); \)

5) The initialisation event must be mapped into a synchronized transition of the process initial activities:
\( \forall t \in T_N \cdot t.src = \top_p \Rightarrow \\
MM.mt(t) = \text{initialisation}. \)

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 \( w \rightarrow w' \) has the corresponding PV state transition \( v \rightarrow v' \) such that \( L(v, w) \land L(v', w') \).

The proof theorem 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 \( \text{receive} \):
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 pursuit 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 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 $\theta, \theta \subseteq C$. For each activity such that its guarantee $G$ permits state update (i.e., $\exists v, v' \cdot v' \not= v \land G(v, v')$), there added a self-transition with guard $\mathcal{T}$ 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

$$\text{new}_{\text{tc} \in \text{id}} \rightarrow \text{last}_{\text{tm} \in \text{id}}$$

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

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:

$$\forall t \cdot (\text{last}_{\text{tm} \in \text{id}} \land \text{WET} \leq \text{exec}_{\text{maxtime}}) \rightarrow \text{Obs}_{1}[\text{c}] < \text{upper}_{\text{bound}}$$

where $\forall$ means "Always, for any execution path", while $\text{Obs}_{1}$ is a special process that starts the clock $\text{Obs}_{1}[\text{c}]$, 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 $\text{upper}_{\text{bound}}$ 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.
Since real-time requirements have a direct impact on 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