Event-B Decomposition Analysis for Systems Behavior Modeling

Kenza Kraibi1, Rahma Ben Ayed1, Joris Rehm2, Simon Collart-Dutilleul1,3, Philippe Bon1,3 and Dorian Petit1,4

1Institut de Recherche Technologique Railenium, F-59300, Famars, France
2CLEARSY, Strasbourg, France
3Univ. Lille Nord de France, IFSTTAR, COSYS, ESTAS, F-59650 Villeneuve d’Ascq, France
4Université Polytechnique Hauts-de-France, LAMIH UMR CNRS 8201, F-59313 Valenciennes, France

Keywords: Formal Methods, Event-B, Refinement, Decomposition, Systems Behavior, Railway Systems.

Abstract: Applications of formal methods to critical systems such as railway systems have been studied by several research works. Their ultimate goal is to increase confidence and to ensure the behavior correctness of these systems. In this paper, we propose to use the Event-B formal method. As a central concept in Event-B, refinement is used to progressively introduce the details of systems requirements, but in most cases, it leads to voluminous and complex models. For this purpose, this paper focuses on decomposition techniques in order to manage the complexity issue in Event-B modeling. It presents a state of the art and an analysis of existing decomposition techniques. Then, an approach will be proposed following this analysis.

1 INTRODUCTION

The analysis and modeling activities of railway dynamic behaviors are major tasks requiring rigorous mechanisms. Based on mathematical foundations, formal methods can help to rigorously carry out these activities and reduce the ambiguity of the specificities of critical systems such as railway signaling systems. The use of formal methods is recommended by the CENELEC 50128 standard (CENELEC, 2011) dedicated to the railway sector. In this paper, we propose to use the Event-B formal method (Abrial et al., 2010; Abrial, 2010) providing appropriate techniques for system modeling based on the B method (Abrial, 1996). B/Event-B methods have been widely used in the railway field in research such as the PERFECT1 and NExTRegio projects (Ben Ayed et al., 2016; Ben Ayed et al., 2014) and in industry sectors as in the METEOR project (Behm et al., 1999). In the same context, CLEARSY2 has also driven railway projects using formal proofs (Sabatier, 2016).

Modeling of critical systems such as railway signaling systems can lead to complex and voluminous models. One of the Event-B techniques for this issue is refinement. Refinement consists in detailing the design to reach a concrete level by progressive steps. However, the final level of modeling is still difficult to manage. In order to reduce this complexity, refinement can be completed by another technique called decomposition of atomicity (Butler, 2009a). Model decomposition is another technique that can reduce the complexity of large models and increase their modularity. This technique consists in dividing a model into sub-models that can be refined separately and more easily than the original one. Several model’s decomposition approaches have been proposed. Some of them are supported by Rodin3 (Butler and Hallerstedt, 2007) plugins4 (Silva et al., 2011).

In this paper, we present in sections 2 and 3, a survey of the existing decomposition techniques. Section 4 describes a railway case study for the existing approaches analysis presented in section 5. Then, this last one illustrates the semantics need of modularity and gives a presentation of the proposed approach and its application to the case study.

---

1PERFECT: http://www.agence-nationale-recherche.fr/Projet-ANR-12-VPTT-0010
2CLEARSY: https://www.clearsy.com/
3Rodin: http://www.event-b.org/
4Modularization: http://wiki.event-b.org/index.php/Modularisation_Plug-in
2 REFINEMENT AND EVENT ATOMICITY DECOMPOSITION

Nowadays, refinement is used to solve complex modeling problems (Badeau and Amelot, 2005). In Event-B, we find two principal types of refinement: data refinement (Back, 1989) and events refinement. Data refinement consists in refining the system state by introducing new variables. It allows the definition of gluing invariants and the invariance properties of the new variables. These invariants allow the correction proof of the refinement (Abrial, 2010). Events refinement consists in introducing new events refining the skip in order to observe the concrete behavior that does not appear in the abstraction (Abrial, 2010). Events refinement allows refining existing events by strengthening their guards and refining their actions.

A proposition in (Butler, 2009a) is called decomposition of event atomicity. This approach is a structuring mechanism for refinement in Event-B. This mechanism is based on decomposing an abstract atomic event to many sub-events, where one event refines this abstract event. Decomposing atomic events is inspired from Jackson System Development (JSD) approach (Butler, 2009b) and it is represented by the ERS approach (Event Refinement Structures) (Dghaym et al., 2016; Dghaym et al., 2017). The idea of the ERS approach is to enrich the Event-B refinement with a graphical tree notation able to represent explicitly the events decomposition in the refinement and the behavior sequencing (Fathabadi et al., 2011). Figure 1 presents a sub-tree. The child nodes of each node are transformed into events in the refinement.

Figure 1: Example of Event Refinement Structures (ERS) diagram.

The nodes order describes the order of events observation (from left to right). XOR specifies the observation of one and only one event. In case of XOR, an event can be refined by many events or any event. AND allows the interleaved execution of events. Despite these refinement virtues, they do not tackle voluminous models issues.

3 MODEL DECOMPOSITION

An Event-B machine can have so many events and state variables that an additional refinement can become difficult to manage. Model decomposition tackles this difficulty by providing a mechanism to divide a large model into several sub-models. For different decomposition techniques (Hoang et al., 2011), descending steps are defined by: modeling the system in an abstract machine, refining the abstract model to fit the structure expected by a given decomposition technique, applying the decomposition, then refining the resulting sub-models independently. Following this guideline, global properties are captured early in the model and guaranteed in the final models by combining refinement and decomposition.

3.1 Shared Event Decomposition

The shared event decomposition is an evolution of decomposition of event atomicity. The author in (Butler, 2009a) proposes this method which makes it possible to separate the variables of a system in two different sub-machines by decomposing a shared event. To decompose a machine using this method, variables to partition in each sub-component are chosen then the decomposition is applied. Generated machines contain the selected variables, and shared events are defined in two different signatures for each sub-machine. These events describe the variables changes.

As illustration, let \( M_0 \) be the machine as in figure 2. Variables \( v1 \) and \( v2 \) of this machine are partitioned respectively in two sub-machines \( M_{1a} \) and \( M_{1b} \). event2 is decomposed in the two sub-components as two events \( \text{event2}' \) and \( \text{event2}'' \), each event describes the change of state applied to \( v1 \) and \( v2 \) respectively.

Figure 2: Decomposition by shared event.
3.2 Shared Variable Decomposition

Abrial proposes in (Abrial and Hallerstede, 2007) the
shared variable decomposition which consists in dis- 
tributing events of a machine between several sub- 
machines. This approach proposes to manage shared 
variables between several events. It is used also for 
decomposing parallel programs (Hoang and Abrial, 
2010). During the machine decomposition, events 
are separated and selected in each sub-machine and 
considered as internal events. A variable that occurs 
only in the internal events is a private variable. If a 
variable is involved in internal events of different sub- 
machines, it is defined in each of them as a shared 
variable that cannot be refined. External events of a 
sub-machine are events that simulate the change of 
state of the external variables in the abstract machine.

Figure 3 illustrates the decomposition by shared variable. The machine $M_0$ is defined by four events 
and it is decomposed into two sub-machines $M_{1a}$ and 
$M_{1b}$, with partitioning its events. Events $\text{event1}$ and $\text{event2}$ (resp. $\text{event3}$ and $\text{event4}$) are internal events 
to the sub-machine $M_{1a}$ (resp. $M_{1b}$). The variable $v_1$ (resp. $v_2$) is private to $M_{1a}$ (resp. $M_{1b}$). As for $v_2$, it 
is a shared variable. Consequently, the machine $M_{1a}$ (resp. $M_{1b}$) contains the external event $\text{event3}'$ (resp. $\text{event2}'$) which simulates the state changes made by 
$\text{event3}$ (resp. $\text{event2}$) on $v_2$ in $M_0$.

![Figure 3: Decomposition by shared variable.](image)

3.3 Other Decomposition Methods and 
Summary

In addition to refinement and decomposition by 
shared variable, generic instantiation is another proposi-
tion of Abrial in (Abrial and Hallerstede, 2007). It is based on the reuse of the abstract model 
with slight modifications by instantiating sets and 
constants of this model. In (Hoang et al., 2011), the 
modularization is another proposition based on defini-
ing interfaces in B method. This approach promotes 
the use of USES clause in order to call operations.

Fragmentation and distribution approach in (Siala 
et al., 2016) defines a specification using DSL (Do-
main Specific Language) (Van Deursen et al., 2000) 
to decompose a model. In the same context, (Hoang 
et al., 2017) propose also a technique based on the 
use of a classical-$B$ clause. This approach proposes 
the use of a composition mechanism based on the use 
of INCLUDES clause. So, the including machine can 
use variables and invariants of the included machine.

Our target is to use specific systems behavior tech-
niques for modeling. In opposition, the previous cited 
approaches rely on the implication of some classical-
$B$ method semantics or the use of another language as 
DSL.

Currently, the railway industry models its systems 
on the basis of a linear modeling. However, the ob-
tained models are voluminous and difficult to man-
age. The aim of our work is, on the one hand, mod-
eling the behavior of railway signaling systems and 
the management of the resulting models complexity 
on the other hand. For this point, we choose to pro-
cede with model decomposition through shared vari-
able decomposition and shared event decomposition.

4 RAILWAY CASE STUDY

In order to analyze the existing approaches and illus-
strate our contribution, we have modeled and formally 
proved a case study of railway signaling systems on 
Atelier $B^5$ and Rodin tools. The aim is to model a 
system which allows the trains control, in other words 
ensure a safe train circulation in a certain railway net-
work containing signals, points, crossings... The main 
goal of this case study is to avoid trains rear-end col-
lisions as in figure 4.

This case study is a simple example that focuses 
on a particular requirement of a railway network and it 
contains relevant elements to the decomposition anal-
ysis. This example is representative of what is done in 
the industrial railway field and in sub-systems traffic 
management such as the European Rail Traffic Man-
age System$^6$ (ERTMS).

![Figure 4: Example of a train rear-end collision.](image)

In a one-way traffic split into blocks $B_i$ as shown 
in figure 5, let consider two trains $Train \ A$ and $Train 

$^5$Atelier B tool: https://www.atelierb.eu/

$^6$European Rail Traffic Management System: http://www.ertms.net
B. Train A follows Train B. The trains are moving by a certain number of steps. The trains movements are based on the position of the front.train and of the end.train of each train. Each block has a front.block and an end.block. Two block states are possible: occupied (red block) or free (green block).

Abstract Machine. In an abstract machine $M_0$ (cf. Figure 6), we define trains movements. $M_0$ defines the variables describing the trains front and the trains end positions (line 2 in figure 6): front.trainA, front.trainB, end.trainA, and end.trainB, and the events that describe the trains movements:

- move_front.trainA: as shown in figure 6, change the position of the Train A front (line 9) without catching up the next train.
- move_end.trainA: change the Train A end position taking into consideration the position of its front.
- move_front.trainB: change the position of the Train B front.
- move_end.trainB: change the position of the Train B end taking into consideration its front position.

MACHINE $M_0$
REFINES

REFINEMENT $M_1$ SEE CONTEXT $C_1$
REFINES $M_0$

properties (axioms) such as the blocks do not intersect, etc. A block can be Free or Occupied.

1. CONTEXT $C$
2. DEFINITIONS Block $\Rightarrow$ NATURAL
3. SETS BlockState $= \{\text{Free},\text{Occupied}\}$;
4. SUBSYS $= \{\text{TRAIN, TRACK}\}$
5. CONSTANTS front.block, end.block, next.block
6. AXIOMS front.block $: \text{Block} \rightarrow \text{NATURAL}$
7. & end.block $: \text{Block} \rightarrow \text{NATURAL}$
8. & next.block $= \%bk.(bk : \text{Block} \mid bk \neq \text{end.block})$
9. & ...
10. END

In machine $M_1$ (cf. Figure 9), the variables that describe the change of the blocks states by block.state and the intermediate variables are defined.

1. REFINEMENT $M_1$ REFINES $M_0$
2. VARIABLES front.trainA, front.trainB, end.trainA, end.trainB, block.state, next.turn,
3. \text{fst.t}A.block, \text{lst.t}A.block, \text{fst.t}B.block, \text{lst.t}B.block
4. INVAR $\forall tAblock \& \text{block.state} = \text{next.block}(\text{fst.t}A.block)$
5. \& \text{end.train}($\text{fst.t}B.block) \leq \text{end.trainB}$
6. \& \text{block.state}(\text{block}) = \text{Free} \Rightarrow \text{bk} \neq \text{lst.t}B.block$
7. & ...$
8. EVENTS enter.tA.block ref move_front.trainA =
9. ANY step
10. WHERE step $: \text{NATURAL}$
11. & block.state(next.block(fst.tA.block)) = \text{Free}$
12. & front.block(fst.tA.block) \leq \text{front.trainA} + \text{step}$
13. & \text{front.trainA} + \text{step} < \text{front.block(next.block(fst.tA.block))}$
14. & next.turn $= \text{TRAIN}$
15. THEN
16. next.turn := \text{TRACK}$
17. front.trainA := \text{front.trainA} + \text{step}$
18. \text{fst.t}A.block := \text{next.block(fst.tA.block)}$
19. \text{END}$
20. ...
21. \text{END}$

Figure 7: Structure of the case study model.

Figure 8: The context defining blocks.

Figure 9: The case study refinement machine.

To avoid a rear-end collision, the position of the Train B end must always be in front of the position of the Train A front. This is specified by the invariant:

$\text{front.trainA} < \text{end.trainB}$

Refinement. In a second phase, we define a more concrete machine introducing the blocks notation and the trains movements on blocks. $M_1$ refining $M_0$ and seeing a context $C$ (cf. figure 7).

The context $C$, as in figure 8, specifies the blocks, their beginnings and ends positions and some track

These intermediate variables allow the communication between the train and the track such as the occupied block by a train (lines 4 in figure 9). Train A can occupy more than one block, so variables $\text{fst.t}A.block$ and $\text{lst.t}A.block$ respectively describe the occupied block by the Train A front and the block occupied by the Train A end. In the same way are de-
fined \texttt{fst\_tBblock} an \texttt{lst\_tBblock} for Train B. The variable \texttt{next\_turn} allows the transition from a \texttt{Track} behavior to a \texttt{Train} behavior and vice versa.

The events of this machine are those defined in $M_0$ refining themselves and other new refining events:

- \texttt{enter\_tAblock} (resp. \texttt{enter\_tBblock}): occupies a block by \texttt{Train} A (resp. \texttt{Train} B) refining \texttt{move\_front\_trainA} (resp. \texttt{move\_front\_trainB}).
- \texttt{free\_tAblock} (resp. \texttt{free\_tBblock}): frees a block by \texttt{Train} A (resp. \texttt{Train} B) refining \texttt{move\_end\_trainA} (resp. \texttt{move\_end\_trainB});
- \texttt{TRACKevent}: a new event changing the block state.

A block can be occupied at most by one train, in other words the occupied block by the end of \texttt{Train} B named \texttt{lst\_tBblock} should always be in front of the occupied block by the front of \texttt{Train} A named \texttt{fst\_tAblock} as defined in the invariant: \texttt{fst\_tAblock} < \texttt{lst\_tBblock}. Another useful invariant is also defined in order to ensure the distance between \texttt{Train} A and \texttt{Train} B: $\forall \ bk. (bk:Block \ & \ next\_turn = \texttt{TRAIN} \ \& \ block \_state(bk) = \texttt{Free} \ \Rightarrow \ bk \neq \texttt{lst\_tBblock})$

Figure 10 shows an example of a possible scenario of trains movements. Using \textit{ProB}\footnote{\textit{ProB}: www3.hhu.de/stups/prob/index.php/Main\_Page} (Leuschel and Butler, 2003), an animation is elaborated on the model.

In a first step, each of \texttt{Train} A and \texttt{Train} B occupy distinct blocks. Then, in step 2 \texttt{Train} B moves to the next block and occupies it. The blue train shadow presents the previous train position. As a third step, \texttt{Train} B moves inside the associated block, \texttt{Train} A releases the previous block. Hence, in step 4 \texttt{Train} A enters the next free block. Note that not all the model elements are presented in the paper\footnote{The case study model and proof in \textit{Atelier B} and \textit{Rodin}, animation in \textit{ProB} and different \textit{Rodin} plugins application can be provided on demand}.

\section{PROPOSED APPROACH}

\subsection{Analysis}

As mentioned in subsection 3.3, this paper focuses on the shared variable decomposition and the shared event decomposition. Both approaches are starting from a machine and decomposing it into two other new machines, then refining the resulting sub-machines separately. Let apply the decomposition by the shared event and by the shared variable plugins. The machine to decompose is $M_1$ and the resulting sub-machines are $M_{2a}$ and $M_{2b}$.

During the \textbf{shared event decomposition}, not all actions are accepted to be decomposed and variables partitioning is not always possible. Table 1 shows different types of actions that make variables states evolve. $v_1$ and $v_2$ are an example of the case study variables.

![Figure 10: An example of trains movement scenario.](image)

<table>
<thead>
<tr>
<th>Actions types of $M_1$</th>
<th>$M_{2a}(v_1)$</th>
<th>$M_{2b}(v_2)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>act1 $v_1 :: (v_1=1)$</td>
<td>$v_1 :: (v_1=1)$</td>
<td>$v_2 :: (v_2=2)$</td>
</tr>
<tr>
<td>act2 $v_1, v_2 :: (v_1=2 \ &amp; \ \ v_2=v_1+2)$</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>act3 $v_1, v_2 :: 1,2$</td>
<td>$v_1 :: 1$</td>
<td>$v_2 :: 2$</td>
</tr>
<tr>
<td>act4 $v_1 :: v_2+1$</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>act5 $v_1 :: {v_2, 1, 2}$</td>
<td>–</td>
<td>–</td>
</tr>
</tbody>
</table>

An error message is displayed asking to simplify the actions; the assignment is too complex because it refers to elements belonging to different sub-components. The obtained errors are shown by this symbol `–`. For the shared event decomposition, predicates (invariants and guards) and actions should not refer to variables that must be partitioned into different sub-components. As an example, the substitution \textit{becomes such that in} act2 \textit{cannot be decomposed}: $v_1, v_2 :: (v_1 = 2 \ \& \ \ v_2 = v_1 + 2)$.

\textbf{For the shared variable decomposition}, the event partition is always possible and can generate sub-components. However, this decomposition may be less relevant because the model to be decomposed contains a large number of shared variables, especially in case of decomposing complex refinements rich with shared variables (Silva et al., 2011) which is the case in the case study. Furthermore, there exists a restriction of this method: shared variables and external events must be present in the resulting sub-components and cannot be refined when refining these sub-components (Abrial, 2009).

For these reasons, it may be necessary to proceed with an intermediate preparation step to resolve complex predicates such as invariants, guards and axioms.
as well as substitutions (actions) by separating the variables assigned to different sub-components. This separation is done by applying an additional manual refinement step before the decomposition (Abrial, 2009). The user must explicitly separate the variables in this refinement by introducing an auxiliary parameter \( p \). For example, the predicate \( v1 = v2 \) becomes \( p = v2 \ & \ v1 = p \). If this manual refinement step is not performed, the complex predicates and substitutions are automatically marked by the tool via a message frame and then the user’s intervention is required to perform the separation explicitly. After this plugins experimentation, we had identified some limitation and issues in the generated machine:

- States changes of several variables in the same action, such as \textit{becomes such that} substitution, cannot be decomposed and should be dealt with the user’s intervention by an intermediate step of refinement and replacing the variables in the predicate with parameters;
- Loss of information when decomposing guards;
- Loss of shared invariant involving shared variables. As long as the resulting sub-machines are not refining the initial machine, the shared invariant is not preserved;
- Generation of empty events in the sub-component;
- Need of an intermediate step of a manual refinement before applying the decomposition.

### 5.2 Discussion

The choice of a decomposition method depends on the work finality:

- Shared variable decomposition can decompose models by functionality, for instance in the railway field, an initial railway signaling model can be decomposed into three sub-components: train integrity, block release and train communication;
- Shared event decomposition is based on partitioning the behavior of a system, e.g. partitioning according to different types of trains movement such as movement under the national Automatic Train Protection (ATP) system or under ERTMS levels.

Nonetheless, the industrial need is to reason on sub-systems, in other words, to take into account both the behavior and the functionality. The use of shared variable decomposition or the shared event decomposition does not address this need. Hence, after this analysis of the existing approaches and according to our industrial needs, some limitations to these techniques are identified, among others, the loss of shared invariants preserving a major safety property. Also, after the generation of the sub-machines by the plugin, the link between the original machine and the sub-machine is not explicit.

### 5.3 Proposed Approach

The analysis above leads us to build a new decomposition approach that corresponds to the industrial need. This technique is based on the decomposition by refinement. By refining the abstract machine while the decomposition, the resulting sub-machines keep the preservation of shared invariants, especially safety invariants. In addition, this approach defines a new semantic link between sub-machines: \textit{REFSEES}. This link allows variables, invariants, constants, sets and properties visibility of a sub-machine by the other sub-machines.

The \textit{REFSEES} is a similar notion to the \textit{SEES} of the \textit{classical-B} with a particular characteristics, The name of the \textit{REFSEES} clause is a combination of \textit{REFINEMENT} and \textit{SEES} which means it allows a refinement machine to see another refinement machine. So we add a clause \textit{REFSEES} to the machine \( M_{1a} \) (resp. \( M_{1b} \), which would make reference to the variables of the machine \textit{seen} \( M_{1b} \) (resp. \( M_{1a} \)). So there, we have a circular dependency with this notion of \textit{REFSEES}. In \textit{classical-B} there is normally no circular dependency and machines cannot see a refinement machine. Contrary to \textit{SEES} clause, \textit{REFSEES} can have a refinement machine as identifier and can be used in a cyclic way.

Figure 11 illustrates the decomposition by refinement of a machine \( M_0 \) into two sub-machines \( M_{1a} \) and \( M_{1b} \):

- \( M_0 \) defines variables \( x, y \) and \( z \), invariants to be preserved \( \mathit{I}(x,y,z) \) and \textit{abstract_event}. An \textit{abstract_event} contains guards \( \mathit{G}(x,y,z) \) and before/after predicates \( \mathit{R}(x,y,z,x',y',z') \).
- The resulting sub-machine \( M_{1a} \) (resp. \( M_{1b} \)) defines the private variable \( x_{1a} \) (resp. \( y_{1a} \)) refining \( x \) (resp. \( y \)). \( z \) is considered as a shared variable which can be refined by \( z_{1a} \) (resp. \( z_{1b} \)) in \( M_{1a} \) (resp. \( M_{1b} \)). Each machine defines gluing invariants \( J_{1a} \) and \( J_{1b} \). \textit{abstract_event} is refined by \textit{reca} in \( M_{1a} \) and by \textit{recb} in \( M_{1b} \). The variable \( x_{1a} \) (resp. \( y_{1b} \)) is visible by \textit{recb} (resp. \textit{reca}) in \( M_{1b} \) (resp. \( M_{1a} \)).

Table 2 is a visibility table of \textit{REFSEES}. Sets and constants of \( M_{1a} \) are visible by axioms, invariants and events of \( M_{1b} \). Private variables of \( M_{1a} \) are only visible by \( M_{1b} \) events. As for shared variables, they are visible and able to be modified by both the sub-machines.

![Image](image-url)
MACHINE M0
VARIABLES x,y,z
INVARIANT S(x,y,z)
EVENTS
abstract event
when G(x,y,z)
then x,y,z : ...

Our aim is that this approach will be used by both Rodin and Atelier B tools. In a future work, we will define

ICSOFT 2019 - 14th International Conference on Software Technologies

5.4 Application of the Approach to the Case Study

The goal is to decompose the machine M1 by separating track and train behaviors and functionalities using the decomposition by refinement as presented in figure 13.

The Track machine, in figure 14, refines M1 and refsees the Train machine through the REFSEES link. It describes the train variables like front_trainA and the trains movement events e.g. enter_fblock. The variable next_turn is a shared variable of Track and Train. Partitioned events keep their guards in the sub-machines. Events that are not needed in a sub-machine are refined such that they cannot be observed anymore such as enter_fblock in the Track machine. In Rodin, this is done automatically since the event is not displayed in the refining machine.

6 CONCLUSION

Modeling railway signaling systems in Event-B produces complex models rich with variables and events. Various techniques have been proposed to cope with this voluminosity issue such as the decomposition technique. Although decomposition approaches promote the modularization of critical systems, some of them do not totally cope with this issue. The analysis of these approaches, on the base of the defined case study, leads to the identification of certain restrictions. As a consequence, we propose in this paper a new approach based on the decomposition by refinement using a new link between sub-machines that addresses the definition of a new semantic. This approach will guarantee the preservation of invariants through the refinement and the visibility of private variables of other sub-machines through the REFSEES link. Our aim is that this approach will be used by both Rodin and Atelier B tools. In a future work, we will define
Multiple levels of refinement

\[ M_{n-1} \]

\[ M_n \]

\[ M_{(n+1)c_1} \]

\[ M_{(n+1)c_2} \]

\[ M_{(n+1)a} \]

Figure 12: Structure of the proposed approach.

Figure 13: Application of the decomposition by refinement on the case study.

Figure 14: Resulting sub-machine Track.

Figure 15: Resulting sub-machine Train.

ACKNOWLEDGMENT

This work is supported by PRECOM (Global Safety Proofs for Modular Design/PR\Euves de Sécurité globale pour la CO\nception Modulaire) as a part of IRT Railenium\textsuperscript{9} projects in collaboration with CLEARSY.

REFERENCES


\textsuperscript{9}Railenium: https://railenium.eu/


