Event-B Decomposition Analysis for Systems Behavior Modeling

Kenza Kraibi[1], Rahma Ben Ayed[1], Joris Rehm[2], Simon Collart-Dutilleul[1,3], Philippe Bon[1,3] and Dorian Petit[1,4]

[1] *Institut de Recherche Technologique Railenium, F-59300, Famars, France*

[2] *CLEARSY, Strasbourg, France*

[3] *Univ. Lille Nord de France, IFSTTAR, COSYS, ESTAS, F-59650 Villeneuve d'Ascq, France*

[4] *Université 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 *PERFECT*[1] 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, CLEARSY[2] 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 *Rodin*[3] (Butler and Hallerstede, 2007) plugins[4] (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.

[1] *PERFECT*: http://www.agence-nationale-recherche.fr/Projet-ANR-12-VPTT-0010

[2] CLEARSY: https://www.clearsy.com/

[3] *Rodin*: http://www.event-b.org/

[4] Modularization: http://wiki.event-b.org/index.php/Modularisation_Plug-in

Kraibi, K., Ben Ayed, R., Rehm, J., Collart-Dutilleul, S., Bon, P. and Petit, D.
Event-B Decomposition Analysis for Systems Behavior Modeling.
DOI: 10.5220/0007929602780286
In *Proceedings of the 14th International Conference on Software Technologies (ICSOFT 2019)*, pages 278-286
ISBN: 978-989-758-379-7

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.

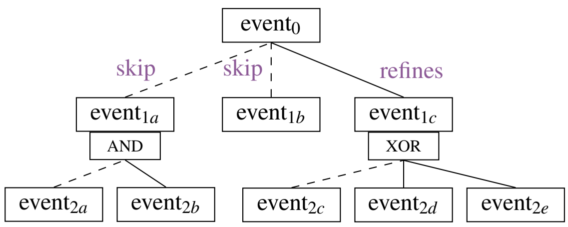

Figure 1: Example of Event Refinement Structures (ERS) diagram.

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 interlaced 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 *event2'* and *event2"*, each event describes the change of state applied to *v1* and *v2* respectively.

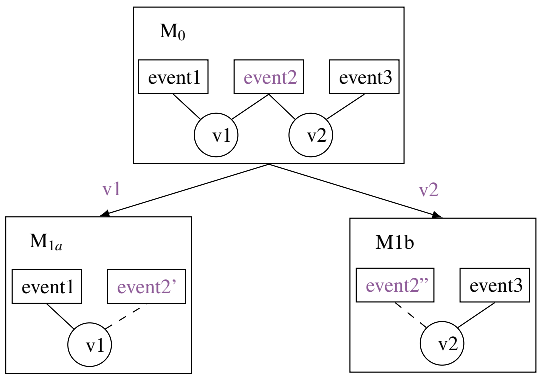

Figure 2: Decomposition by shared event.

3.2 Shared Variable Decomposition

Abrial proposes in (Abrial and Hallerstede, 2007) the shared variable decomposition which consists in distributing 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 to be separated are 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} by partitioning its events. Events *event1* and *event2* (resp. *event3* and *event4*) are internal events to the sub-machine M_{1a} (resp. M_{1b}). The variable *v1* (resp. *v2*) is private to M_{1a} (resp. M_{1b}). As for *v2*, it is a shared variable. Consequently, the machine M_{1a} (resp. M_{1b}) contains the external event *event3'* (resp. *event2'*) which simulates the state changes made by *event3* (resp. *event2*) on *v2* in M_0.

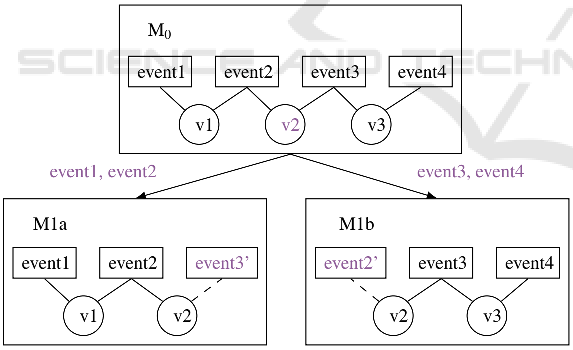

Figure 3: Decomposition by shared variable.

3.3 Other Decomposition Methods and Summary

In addition to refinement and decomposition by shared variable, generic instantiation is another proposition 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 defining 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 (Domain 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 techniques 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 obtained models are voluminous and difficult to manage. The aim of our work is, on the one hand, modeling the behavior of railway signaling systems and the management of the resulting models complexity on the other hand. For this point, we choose to proceed with model decomposition through shared variable decomposition and shared event decomposition.

4 RAILWAY CASE STUDY

In order to analyze the existing approaches and illustrate 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 network containing signals, points, crossings... The main goal of this case study is to avoid trains rear-end collisions 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 analysis. 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 Management System*[6] (ERTMS).


Figure 4: Example of a train rear-end collision.

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).

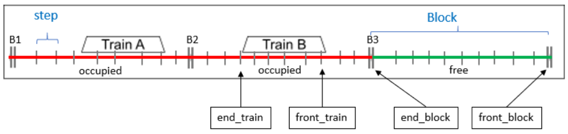

Figure 5: Case study description.

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 *Train B* front position.
- move_end_trainB: change the position of the *Train B* end taking into consideration its front position.

```
1.  MACHINE M0
2.  VARIABLES front_trainA, front_trainB, end_trainA,
            end_trainB
3.  INVARIANTS   front_trainA < end_trainB  & ...
4.  EVENTS
5.  move_front_trainA =
6.     ANY    step
7.     WHERE    step : NATURAL1
8.        & front_trainA + step < end_trainB
9.     THEN    front_trainA := front_train1 + step
10.    END;
11.    ....
12.    END
```

Figure 6: Abstract machine excerpt of the case study.

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:

$$front\_trainA < end\_trainB$$

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

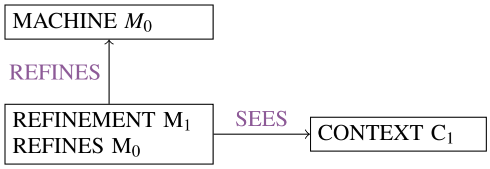

Figure 7: Structure of the case study model.

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

```
1.  CONTEXT    C
2.  DEFINITIONS    Block == NATURAL
3.  SETS    BlockState = {Free,Occupied};
4.          SUBSYS = {TRAIN, TRACK}
5.  CONSTANTS    front_block, end_block, next_block
6.  AXIOMS    front_block : Block → NATURAL
7.            & end_block : Block → NATURAL
8.            & next_block = %bk.(bk : Block | bk+1)
9.            & ...
10. END
```

Figure 8: The context defining blocks.

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 M1 REFINES M0
2.  VARIABLES front_trainA, front_trainB, end_trainA,
3.    end_trainB, block_state, next_turn,
4.    fst_tAblock, lst_tAblock, fst_tBblock, lst_tBblock
5.  INVARIANTS fst_tAblock < lst_tBblock
6.    & end_train(lst_tBblock) ≤ end_trainB
7.    & ∀bk.(bk : Block & next_turn = TRAIN&
8.    block_state(bk) = Free ⇒ bk ≠ lst_tBblock)
9.    & ...
10.  EVENTS
11.  enter_tAblock ref move_front_trainA =
12.     ANY    step
13.     WHERE    step : NATURAL1
14.        & block_state(next_block(fst_tAblock)) = Free
15.        & front_block(fst_tAblock)≤ front_trainA+step
16.        & front_trainA + step <
22.                front_block(next_block(fst_tAblock))
17.        & next_turn = TRAIN
18.     THEN
19.        next_turn := TRACK
20.        || front_trainA := front_trainA + step
21.        || fst_tAblock := next_block(fst_tAblock)
22.     END;
23.  ...
24.  END
```

Figure 9: The case study refinement machine.

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 *fst_tAblock* and *lst_tAblock* respectively describe the occupied block by the *Train A* front and the block occupied by the *Train A* end. In the same way are defined *fst_tBblock* an *lst_tBblock* for *Train B*. The variable *next_turn* allows the transition from a *Track* behavior to a *Train* behavior and vice versa.

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

- enter_tAblock (resp. enter_tBblock): occupies a block by *Train A* (resp. *Train B*) refining move_front_trainA (resp. move_front_trainB).
- free_tAblock (resp. free_tBblock): frees a block by *Train A* (resp. *Train B*) refining move_end_trainA (resp. move_end_trainB);
- 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 *Train B* named *lst_tBblock* should always be in front of the occupied block by the front of *Train A* named *fst_tAblock* as defined in the invariant: $fst\_tAblock < lst\_tBblock$. Another useful invariant is also defined in order to ensure the distance between *Train A* and *Train B*: ∀ bk.(bk:Block & next_turn = TRAIN & block_state(bk) = Free ⇒ bk ≠ lst_tBblock)

Figure 10 shows an example of a possible scenario of trains movements. Using *ProB*[7] (Leuschel and Butler, 2003), an animation is elaborated on the model.

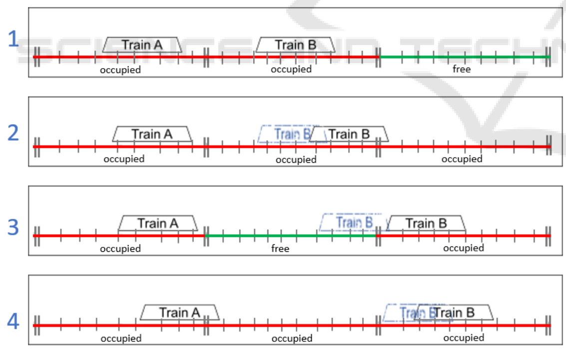

Figure 10: An example of trains movement scenario.

In a first step, each of *Train A* and *Train B* occupy distinct blocks. Then, in step 2 *Train B* moves to the next block and occupies it. The blue train shadow presents the previous train position. As a third step, while *Train A* is moving inside the associated block, *Train B* releases the previous block. Hence, in step 4 *Train A* enters the next free block. Note that not all the model elements are presented in the paper[8].

[7] *ProB: www3.hhu.de/stups/prob/index.php/Main_Page*

[8] The case study model and proof in *Atelier B* and *Rodin*, animation in *ProB* and different *Rodin* plugins application can be provided on demand.

5 PROPOSED APPROACH

5.1 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 **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. *v1* and *v2* are an example of the case study variables.

Table 1: Application of the shared event decomposition.

Actions types of M_1		$M_{2a}(v1)$	$M_{2b}(v2)$
act1	v1 :| (v1=1) || v2 :| (v2=2)	v1 :| (v1=1)	v2 :| (v2=2)
act2	v1,v2 :| (v1=2 ∧ v2=v1+2)	–	–
act3	v1,v2 := 1,2	v1 := 1	v2 := 2
act4	v1 := v2+1	–	–
act5	v1 :: {v2,1,2}	–	–

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 *becomes such that* in *act2* cannot be decomposed: $v1, v2 :| (v1 = 2 \wedge v2 = v1 + 2)$

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 *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: *REFSSES*. This link allows variables, invariants, constants, sets and properties visibility of a sub-machine by the other sub-machines.

The *REFSSES* is a similar notion to the *SEES* of the *classical-B* with a particular characteristics, The name of the *REFSEES* clause is a combination of *REFINEMENT* and *SEES* which means it allows a refinement machine to see another refinement machine. So we add a clause *REFSEES* to the machine M_{1a} (resp. M_{1b}), which would make reference to the variables of the machine seen M_{1b} (resp. M_{1a}). So there, we have a circular dependency with this notion of *REFSEES*. In *classical-B* there is normally no circular dependency and machines cannot *see* a refinement machine. Contrary to *SEES* clause, *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 I(x,y,z) and abstract_event. An abstract_event contains guards *G(x,y,z)* and before/after predicates *R(x,y,z,x',y',z')*.
- The resulting sub-machine M_{1a} (resp. M_{1b}) defines the private variable x_{1a} (resp. y_{1b}) 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}. abstract_event is refined by re_a in M_{1a} and by re_b in M_{1b}. The variable x_{1a} (resp. y_{1b}) is visible by re_b (resp. re_{1a}) in M_{1b} (resp. M_{1a}).

Table 2 is a visibility table of *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.

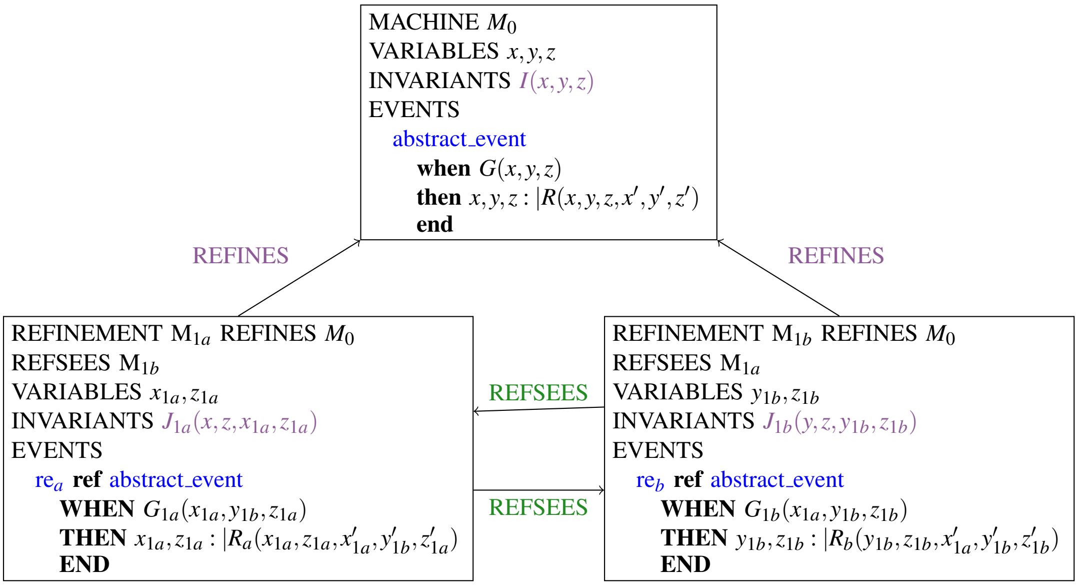

Figure 11: Decomposition by refinement on two sub-machines.

Table 2: REFSEES visibility of M_{1a} by M_{1b}.

M_{1a} \ M_{1b}	AXIOMS	INV	INIT / EVENTS
Sets	visible	visible	visible
Constants	visible	visible	visible
Private variables			visible
Shared variables			visible
Events			

Figure 12 shows the general structure of the proposed approach. The decomposition can be applied on a certain level of refinement and done by multiple horizontal refinements. As shown in the figure a machine M_{n-1} can be refined by m machines. These resulting sub-machines keep the refinement link with the root.

5.4 Application of the Approach to the Case Study

The goal is to decompose the machine M_1 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 M_1 and *refsees* the *Train* machine through the *REFSEES* link. *Track* contains the variables associated to the track like the blocks states variable: *block_state*. It contains also events that make these variables evolve such as *TRACKevent*.

As for the *Train* machine, in figure 15, it refines M_1 and *refsees* the *Track* machine through the *REFSEES* link. It describes the train variables like *front_trainA* and the trains movement events e.g. *enter_tAblock*. 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_tAblock* 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

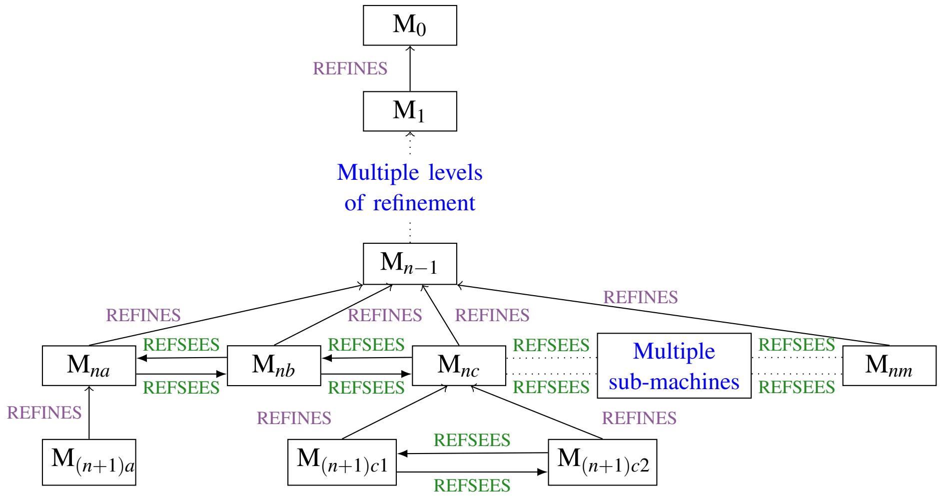

Figure 12: Structure of the proposed approach.

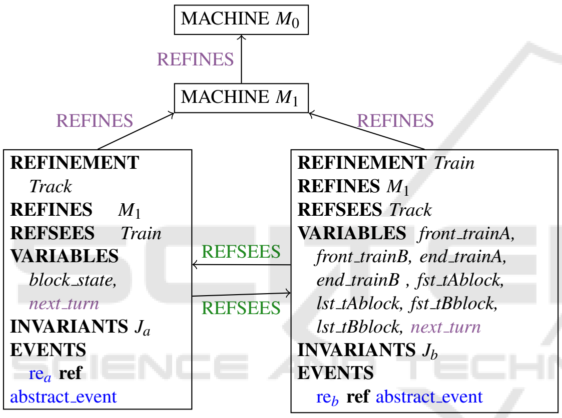

Figure 13: Application of the decomposition by refinement on the case study.

```
INVARIANTS
    block_state : Block → BlockState
    & next_turn : SUBSYS & ...
EVENTS
    TRACKevent =
    SELECT next_turn = TRACK
    THEN
        next_turn:=TRAIN || block_state:=(Block*Free)
    <+ ((lst_tAblock..fst_tAblock U lst_tAblock ..
    fst_tAblock) *Occupied)
    END;
    enter_tAblock=SELECT 0=1 THEN skip END;
...
END
```

Figure 14: Resulting sub-machine *Track*.

proof obligations formulas for these new notations. As consequence, for each sub-machine proof obligations are generated taking into account the link to the other sub-machines.

```
INVARIANTS next_turn : SUBSYS
    & fst_tAblock : Block  & lst_tAblock : Block
    & fst_tBblock : Block & lst_tBblock : Block
EVENTS
    enter_tAblock
    ANY     step
    WHERE    step : NATURAL1
    & block_state(next_block(fst_tAblock)) = Free
    & front_block(fst_tAblock)≤ front_trainA+step
    & front_trainA+step<
      front_block(next_block(fst_tAblock))
    & next_turn = TRAIN
    THEN        next_turn := TRACK
      || front_trainA := front_trainA + step
      || fst_tAblock := next_block(fst_tAblock)
    END; ...
    TRACKevent = SELECT 0=1 THEN skip END
END
```

Figure 15: Resulting sub-machine *Train*.

ACKNOWLEDGMENT

This work is supported by **PRESCOM** (Global Safety Proofs for Modular Design/**PRE**uves de **S**écurité globale pour la **CO**nception **M**odulaire) as a part of *IRT Railenium*[9] projects in collaboration with CLEARSY.

REFERENCES

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

[9]*Railenium*: https://railenium.eu/

Abrial, J.-R. (2009). Event Model Decomposition. *Technical report/[ETH, Department of Computer Science*, 626.

Abrial, J.-R. (2010). *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, New York, NY, USA.

Abrial, J.-R., Butler, M., Hallerstede, S., Hoang, T. S., Mehta, F., and Voisin, L. (2010). Rodin: an open toolset for modelling and reasoning in Event-B. *International journal on software tools for technology transfer*, 12(6):447–466.

Abrial, J.-R. and Hallerstede, S. (2007). Refinement, Decomposition, and Instantiation of Discrete Models: Application to Event-B. *Fundamenta Informaticae*, 77(1-2):1–28.

Back, R.-J. (1989). Refinement Calculus, Part II: Parallel and Reactive Programs. In *Workshop/School/Symposium of the REX Project (Research and Education in Concurrent Systems)*, pages 67–93. Springer.

Badeau, F. and Amelot, A. (2005). Using B as a high level programming language in an industrial project: Roissy VAL. In *International Conference of B and Z Users*, pages 334–354. Springer.

Behm, P., Benoit, P., Faivre, A., and Meynadier, J. M. (1999). Météor: A Successful Application of B in a Large Project. In Wing, J., Woodcock, J., and Davies, J., editors, *FM'99 - Formal Methods*, volume 1708 of *Lecture Notes in Computer Science*, pages 369–387. Springer Berlin Heidelberg.

Ben Ayed, R., Collart-Dutilleul, S., Bon, P., Idani, A., and Ledru, Y. (2014). B Formal Validation of ERTMS/ETCS Railway Operating Rules. In *International Conference on Abstract State Machines, Alloy, B, TLA, VDM, and Z*, pages 124–129. Springer.

Ben Ayed, R., Collart-Dutilleul, S., and Prun, E. (2016). "Formal Methods To Tailored Solution For Single Track Low Traffic French Lines". In *International Railway Safety Council (IRSC), Paris, France*.

Butler, M. (2009a). Decomposition Structures for Event-B. In *International Conference on Integrated Formal Methods*, pages 20–38. Springer.

Butler, M. (2009b). Incremental Design of Distributed Systems with Event-B. *Engineering Methods and Tools for Software Safety and Security*, 22(131).

Butler, M. and Hallerstede, S. (2007). The Rodin formal modelling tool. In *BCS-FACS Christmas 2007 Meeting-Formal Methods In Industry, London.*

CENELEC, E. (2011). 50128. *Railway applications-Communication, Signaling and Processing Systems-Software for Railway Control and Protection Systems*.

Dghaym, D., Butler, M., and Fathabadi, A. S. (2017). Extending ERS for Modelling Dynamic Workflows in Event-B. In *Engineering of Complex Computer Systems (ICECCS), 2017 22nd International Conference on*, pages 20–29. IEEE.

Dghaym, D., Trindade, M. G., Butler, M., and Fathabadi, A. S. (2016). A Graphical Tool for Event Refinement Structures in Event-B. In *International Conference on Abstract State Machines, Alloy, B, TLA, VDM, and Z*, pages 269–274. Springer.

Fathabadi, A. S., Rezazadeh, A., and Butler, M. (2011). Applying Atomicity and Model Decomposition to a Space Craft System in Event-B. In *NASA Formal Methods Symposium*, pages 328–342. Springer.

Hoang, T. S. and Abrial, J.-R. (2010). Event-b decomposition for parallel programs. In *International Conference on Abstract State Machines, Alloy, B and Z*, pages 319–333. Springer.

Hoang, T. S., Dghaym, D., Snook, C., and Butler, M. (2017). A composition mechanism for refinement-based methods. In *2017 22nd International Conference on Engineering of Complex Computer Systems (ICECCS)*, pages 100–109. IEEE.

Hoang, T. S., Iliasov, A., Silva, R. A., and Wei, W. (2011). A Survey on Event-B Decomposition. *Electronic Communications of the EASST*, 46.

Leuschel, M. and Butler, M. (2003). ProB: A Model Checker for B. In *FME*, volume 2805, pages 855–874. Springer.

Sabatier, D. (2016). Using formal proof and B method at system level for industrial projects. In *International Conference on Reliability, Safety and Security of Railway Systems*, pages 20–31. Springer.

Siala, B., Tahar Bhiri, M., Bodeveix, J.-P., and Filali, M. (2016). Un processus de Développement Event-B pour des Applications Distribuées. *Université de Franche-Comté*.

Silva, R., Pascal, C., Hoang, T. S., and Butler, M. (2011). Decomposition tool for Event-B. *Software: Practice and Experience*, 41(2):199–208.

Van Deursen, A., Klint, P., and Visser, J. (2000). Domain-specific languages: An annotated bibliography. *ACM Sigplan Notices*, 35(6):26–36.