Visual Animation of B Specifications Using Executable DSLs

Asfand Yar
asfand.yar@univ-grenoble-alpes.fr
Univ. Grenoble Alpes, CNRS, Grenoble INP, LIG
Grenoble, France

Akram Idani
akram.idani@univ-grenoble-alpes.fr
Univ. Grenoble Alpes, CNRS, Grenoble INP, LIG
Grenoble, France

Yves Ledru
yves.ledru@univ-grenoble-alpes.fr
Univ. Grenoble Alpes, CNRS, Grenoble INP, LIG
Grenoble, France

Simon Collart-Dutilleul
simon.collart-dutilleul@univ-eiffel.fr
Univ. Gustave Eiffel, Univ. de Lille
Villeneuve d'Ascq, France

ABSTRACT

Visual animation of formal specifications is useful for validation because it facilitates in an explicit illustrative way to show that the specifications satisfy the user's perception of requirements. The technique is especially useful for domain experts who would not be expected to understand formal specifications. However, in most tools, the development of a visual animation is done by formal methods engineers and requires skills in various technologies (*e.g.* Flash, JavaScript, SVG). Our work contributes toward the tools that are dedicated to the B method, such as B-Motion Studio, VisB, etc. In this paper, we show how visual animation can be done using a domain-specific language (DSL), which is expected to be used by domain experts themselves. The advantage is that the mapping between the DSL and the formal specification is written in B itself. The proposed approach is supported by Meeduse, a language workbench built on ProB, an animator and model-checker of the B method.

CCS CONCEPTS

• **Software and its engineering → Domain specific languages**; **Software verification and validation**.

KEYWORDS

B Method, Animation, MDE, DSLs

ACM Reference Format:
Asfand Yar, Akram Idani, Yves Ledru, and Simon Collart-Dutilleul. 2022. Visual Animation of B Specifications Using Executable DSLs. In *ACM/IEEE 25th International Conference on Model Driven Engineering Languages and Systems (MODELS '22 Companion), October 23–28, 2022, Montreal, QC, Canada.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3550356.3561585

1 INTRODUCTION

Validation is essential because misunderstanding the user requirements may lead to erroneous implementations. This task is known to be complex especially for formal specifications because of their mathematical notations. In fact, these notations are difficult to understand by stakeholders who are not trained in formal methods, such as domain experts. To address validation issues, several formal tools [8, 13, 21] provide graphical animation techniques. The intention is to exhibit a visual representation of data and behaviors of a formal specification, making it more readable for domain experts. Visual animation is hence an interesting way to inspect whether formal specifications meet the user requirements.

This paper addresses the graphical animation and visualization of B specifications and provides a new approach complementing tools such as BRAMA [18], AnimB[1], B-Motion Studio [8], B-Motion Web [9], and VisB [22]. Existing tools have shown their strengths in practice, but still some concerns remain, which motivates the current work. First, visual animation often requires specific skills such as in scripting languages (Flash, JavaScript, node.js), or in Scalable Vector Graphics (SVG files), etc. These technologies are not necessarily mastered by the formal methods expert and can be cumbersome to learn and use. In [7], Krings and Körner observed that "*When errors occur, it is not clear in which layer the cause is located in: is it an error in the B model? Is an SVG file broken? Is the config file incorrect? Is there a bug in the JavaScript code? As some errors are not reported, development can be cumbersome if one is not an expert in all technologies*". Second, mapping a specification to a domain representation is time-consuming and maybe error prone. Often this is done by a formal methods expert who is trying to document his own specification. Unfortunately, if the process of validation reveals some misunderstandings, not only the formal specifications must be re-worked, but also the visual representations and the underlying mapping. Furthermore, the formal methods expert might not be familiar with the domain-specific notations.

To deal with the aforementioned concerns, we propose a new technique built on domain-specific languages (DSLs). In our approach, the domain-specific representations are designed using a DSL tool that is created in EMF [19], a well-known MDE platform. The challenge is therefore to make the bridge between the DSL and the formal specification. To this purpose, we use Meeduse [4, 5], the only existing language workbench today that allows both formal reasoning about the correctness of a DSL and the execution of the underlying formal semantics. Meeduse takes advantage of B specifications to define the semantics of a DSL and embeds ProB [11] for the execution capabilities. Our approach favors the separation of

[1] AnimB: https://wiki.event-b.org/index.php/AnimB

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
MODELS '22 Companion, October 23–28, 2022, Montreal, QC, Canada
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-9467-3/22/10…$15.00
https://doi.org/10.1145/3550356.3561585

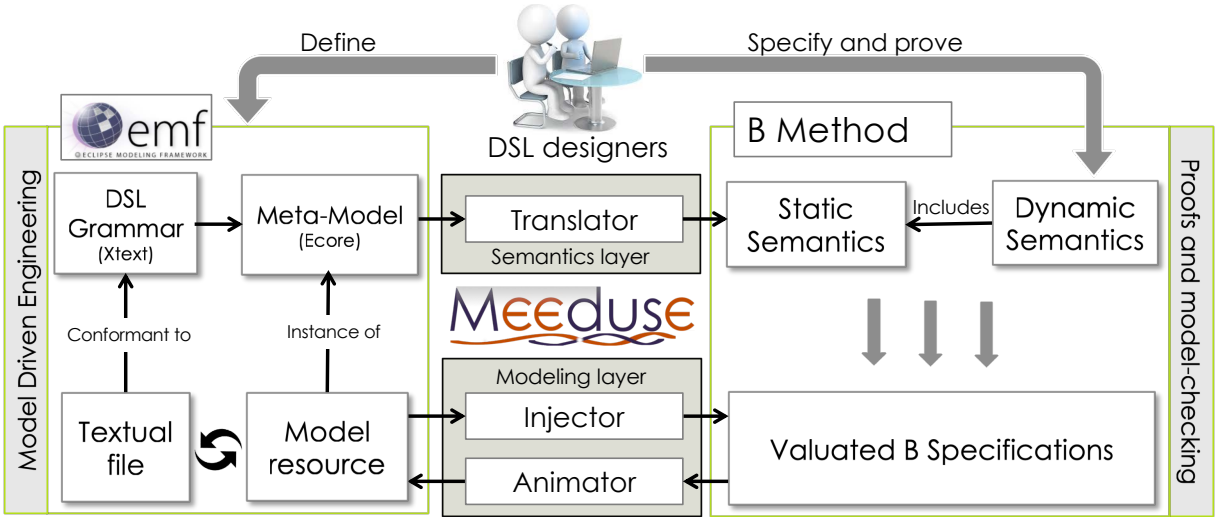

Figure 1: Main principles of Meeduse (taken from [4])

concerns principle: the formal methods expert is responsible for the development of B specifications and the domain expert provides the useful input models thanks to the DSL tool. However, our approach involves another actor, the MDE expert who is responsible for the development of the DSL tool.

Section 2 presents and illustrates via a simple example a quick overview of the main concepts of our approach. Then, Section 3 shows how a domain-centric visual animation can be instrumented in Meeduse. Application to other examples is presented in Section 4. Section 5 discusses the related work. Finally, Section 6 draws the conclusion and the perspectives of this paper.

2 OVERALL APPROACH

2.1 A Simple Example

Leuschel *et* al., in [12] used many examples for the visualization of B specifications. To illustrate our approach we select the Lift example which is well-known in formal verification and validation [16]. Several rules about the specification have been proved, for example, the lift may not move with doors open. The structural part of this specification is given in Figure 2 (named $\text{Lift}_{\text{existing}}$).

It represents one lift that moves between the ground floor (constant groundf) and the top floor (constant topf). Both groundf and topf are natural numbers, such that groundf is less than topf. Variable cur_floor gives the current position of the lift among the floors. Variables direction_up and door_open are booleans, representing whether the direction of the lift is up and whether the door of the lift is open respectively. The call buttons are represented with variable call_buttons, a set of natural numbers, without any distinction between the internal and external buttons of the lift cabin. The specification contains seven B operations that are not presented here for space reasons: move_up, move_down, reverse_lift_up, reverse_lift_down, open_door, close_door and push_call_button. We choose the lift specification because it is a pedagogical example

MACHINE
  $Lift_{\text{existing}}$
CONCRETE_CONSTANTS
  groundf, *topf*
PROPERTIES
  $topf \in \text{NAT} \wedge groundf \in \text{NAT} \wedge (groundf < topf)$
CONCRETE_VARIABLES
  call_buttons, *cur_floor*, *direction_up*, *door_open*
INVARIANT
  $cur\_floor \in (groundf \,..\, topf)$
  $\wedge\ door\_open \in \textbf{BOOL}$
  $\wedge\ call\_buttons \in \text{Pow}(groundf \,..\, topf)$
  $\wedge\ direction\_up \in \textbf{BOOL}$
  $\wedge\ (door\_open = \textbf{TRUE} \Rightarrow cur\_floor \in call\_buttons)$
INITIALISATION
  $cur\_floor := \text{groundf}$
  $||\ door\_open := \textbf{FALSE}$
  $||\ call\_buttons := \emptyset$
  $||\ direction\_up := \textbf{TRUE}$

Figure 2: Lift example [12]

used in the B-Book [1] and referenced in several papers [12, 22]. Furthermore, the example gathers several B data structures allowing us to illustrate various concepts of our approach.

2.2 Meeduse

Our domain-centric approach for visual animation builds on the Meeduse[2] language workbench [5]. The tool is dedicated to the formal instrumentation of domain-specific languages using the B-Method, which allows automated reasoning about their correctness. It embeds ProB [11] to ensure the animation and the verification of domain models. Figure 1 shows the main principles of the tool;

[2]http://vasco.imag.fr/tools/meeduse/

for more details, we refer the reader to [4, 5]. Roughly speaking, the Meeduse approach addresses two layers: the semantics layer (top side of Figure 1) and the modeling layer (bottom side of Figure 1). In the semantics layer, the meta-model of a DSL is translated into B using a classical UML-to-B approach. This step leads to a functional B machine defining the static semantics of the DSL. Regarding the modeling layer, the Meeduse approach follows two steps: inject and animate; and benefits from the ProB tool. Starting from a given model resource conforming to the DSL meta-model, Meeduse creates a valuated B machine that is semantically equivalent to the input model resource. This machine is an extraction of the functional B model in which Meeduse injects valuations to populate the various B data structures. Having this valuated machine, ProB is applied to animate B operations of any B machine that includes the valuated model. It is called "Dynamic Semantics" because it confers to the DSL a behavioral character. At every animation step, when ProB modifies the internal state of the valuated functional model, Meeduse translates back this modification to the input model resource, which results in an automatic animation of the model resource.

2.3 Proposed Architecture

Meeduse requires to provide a B specification of the dynamic semantics. We propose to use an existing one, that is already proved correct and provided by B method experts. Our approach is illustrated in Figure 3 which redefines the dynamic semantics part from Figure 1.

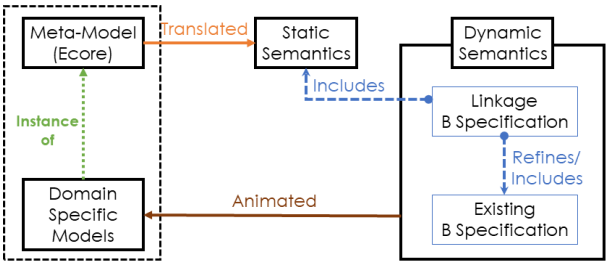

Figure 3: Proposed approach

Since the static semantics (B_{static}) of the DSL is written in B, therefore the remaining piece is to link an existing B specification (B_{existing}) to B_{static}. To this purpose we add a linkage B specification (B_{linkage}) along with B_{existing}. The linkage B specification (B_{linkage}) maps the concepts of B_{existing} to B_{static}. We propose two techniques: Refinement/Inclusion and Inclusion/Inclusion. In the Refinement/Inclusion technique, B_{linkage} refines B_{existing} and includes B_{static}. In the Inclusion/Inclusion technique, B_{linkage} includes both B_{existing} and B_{static}.

Note that B_{linkage} applies B invariants to map the variables of B_{static} to those issued from B_{existing}. By animating B_{linkage}, the states of both B_{existing} and B_{static} are modified and ProB verifies that the mapping is preserved all along the animation. The task of Meeduse is to update the input model resource during the animation conforming to the state of B_{static}, which produces the expected visual animation.

2.4 Illustration

Figure 4 shows a screenshot of Meeduse while animating an EMF-based DSL that represents the Lift example. The domain representation on the right hand side of the figure can be realized by a domain expert using the modelling tool. The latter allows one to define buildings with several lifts. In this illustration, the model editor is developed with Sirius [3], an EMF-based framework dedicated to the design of graphical syntaxes of DSLs. The other views are for interactive animation. The execution view shows the candidate operations computed by ProB from B_{linkage} and the state view shows the current valuations of the various B variables. The execution history records the operation calls that led to the current state. After every animation step, the model resource is updated by Meeduse and Sirius automatically renders the graphical model without the need of any specific implementation effort.

3 DOMAIN-CENTRIC VISUAL ANIMATION

The previous section presented and illustrated a quick overview of the main concepts of our approach. In this section we show, using the Lift example, how a domain-centric visual animation can be instrumented in Meeduse. Note that all the artifacts of this paper are publicly available in the Meeduse git repository[3].

3.1 The Lift DSL

Figure 5 gives the meta-model of our Lift DSL. The three main concepts of the DSL are buildings (class Building), floors (class Floor) and lifts (class Lift). Each floor has at most one upper floor (association up) and one lower floor (association down). Association liftPosition gives the position of a given lift, and selectedFloors refers to the set of selected floors from a given lift. Each floor is composed of zero or many cabins (association cabins), by "cabin" we mean the interface for a lift at a given floor. The concept of cabin is useful for our DSL because it manages the graphical representation: the visual aspect of a cabin changes depending on the current position of the lift and the door state. Class Lift has an attribute Direction that can be either Up or Down. Finally, attribute Door, represents the door, which can be either Closed or Open. The model presented on the right side of Figure 4 is an instance of this meta-model. It gathers two lifts (Lift1 and Lift2) and five floors (Floor1, Floor2, Floor3, Floor4, and Floor5). Each floor has two cabins, corresponding to the two lifts. The arrow symbols at the bottom of the model show the direction of the lift. The direction of Lift1 is up while the direction of the Lift2 is down. The doors of both lifts are closed. The active button beside each door shows the selected floor for a given lift.

Note that the definition of the DSL is done independently from the existing B specification that one would like to visualize. The objective is to focus on the domain concepts and their relationships and build a DSL modeler that is useful for domain experts. For example, there is no explicit notion of call buttons. This concept is somehow similar to the association selectedFloors from Lift to Floor.

[3] https://github.com/meeduse/Samples/tree/main/Lift.

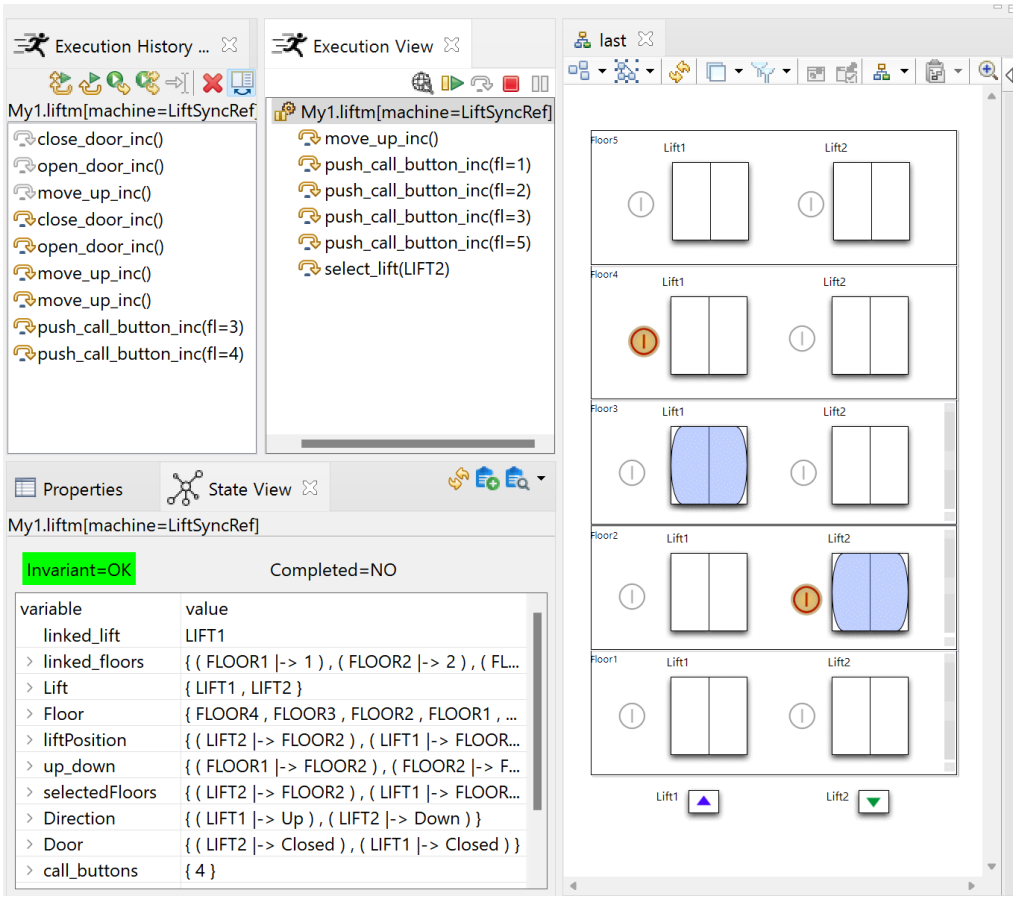

Figure 4: Visual animation in Meeduse

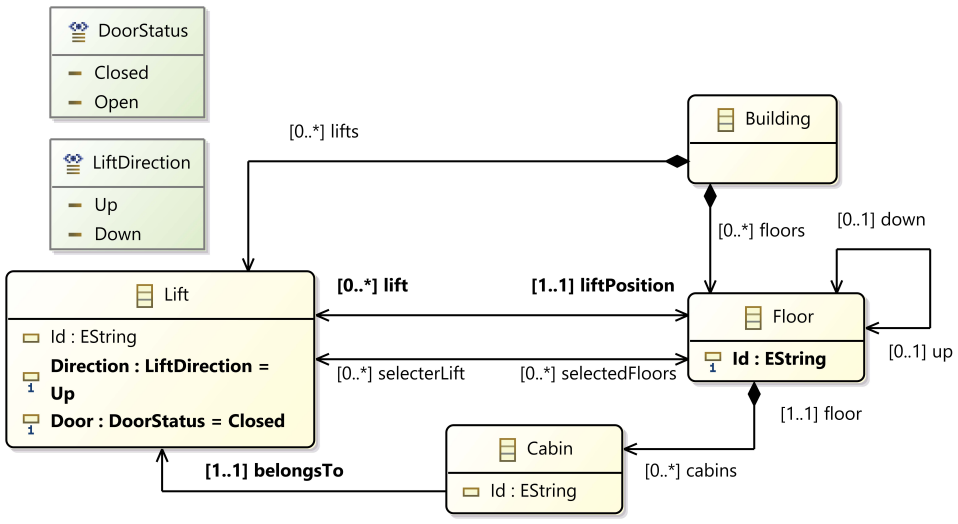

Figure 5: Lift Meta-Model

3.2 Static Semantics

The extraction of B_{static} is done by Meeduse, resulting in a B machine that covers the structure of the meta-model and gives several basic operations such as constructors, destructors, getters, and setters. Figure 6 gives the structural part of this machine (named $Lift_{static}$).

MACHINE
  $Lift_{static}$
SETS
  LiftDirection = {*Up*,*Down*};
  DoorStatus = {*Closed*,*Open*};
  LIFT; *FLOOR*
VARIABLES
  Lift,
  Floor,
  liftPosition,
  up_down,
  selectedFloors,
  Direction,
  Door
INVARIANT
  $Lift \in \text{Pow}(LIFT) \wedge$
  $Floor \in \text{Pow}(FLOOR) \wedge$
  $liftPosition \in Lift \rightarrow Floor \wedge$
  $up\_down \in Floor \rightarrowtail\!\!\!\!\!\rightarrow Floor \wedge$
  $selectedFloors \in Lift \leftrightarrow Floor \wedge$
  $Direction \in Lift \rightarrow LiftDirection \wedge$
  $Door \in Lift \rightarrow DoorStatus$

Figure 6: Structural part of machine $Lift_{static}$

In the generated B specification, we can notice that there are no data structures generated for the Building and Cabin classes. These concepts are useful for the lift DSL but we do not require them in our B specification. In fact, Meeduse features an annotation mechanism that allows the user to select the concepts to be translated. We generated seven variables together with their typing invariants: Lift, Floor, liftPosition, up_down, selectedFloors, Direction, and Door. The operations of $Lift_{static}$ are not included here due to the limitation of space.

3.3 Linking B Data Structures

Figures 2 and 6 gave respectively the existing machine ($Lift_{existing}$) and the static semantics of the lift DSL ($Lift_{static}$). In order to apply Meeduse for visual animation, one needs to create a third machine ($Lift_{linkage}$) that maps B data of both machines.

3.3.1 Mapping a Class to a Machine. Machine $Lift_{existing}$ contains invariants and B operations to move a lift between the floors safely without any error. In machine $Lift_{static}$ multiple objects of class Lift can be created (abstract set LIFT and variable Lift). In this case, we consider that $Lift_{existing}$ will control only one instance of class Lift. Thus, we introduce in machine $Lift_{linkage}$ variable *linked_lift* to select the lift object that will be controlled by the existing machine:

Listing 1: EClass to BMachine

VARIABLE
  linked_lift
INVARIANT
  $linked\_lift \in Lift$
INITIALISATION
  $linked\_lift :\in Lift$

3.3.2 Mapping a Class to a Set. In $Lift_{existing}$ floors are represented with a set of natural numbers ($groundf \,..\, topf$), and in $Lift_{static}$ they are defined with a dedicated variable representing class Floor. The mapping is then done as follows:

Listing 2: EClass to BSet

VARIABLE
  linked_floors
INVARIANT
  $linked\_floors \in Floor \rightarrowtail\!\!\!\!\!\rightarrow groundf \,..\, topf$
INITIALISATION
1. **ANY** *mapFloors* **WHERE**
2. $mapFloors \in Floor \rightarrowtail\!\!\!\!\!\rightarrow groundf \,..\, topf \wedge$
3. $\forall (f1,f2).(f1 \in Floor \wedge f2 \in Floor \wedge (f1 \mapsto f2)$
4. $\in up\_down \Rightarrow mapFloors(f2) = mapFloors(f1) + 1)$
5. **THEN**
6. $linked\_floors := mapFloors$
7. **END**

Relation *linked_floors* is a total bijection meaning that every floor in the DSL must be linked to one value from ($groundf \,..\, topf$) and vice-versa. The initialisation is done such that if a floor f2 is associated to a floor f1 via relation up_down then the value of mapFloors(f2) is equal to mapFloors(f1) plus one.

3.3.3 Mapping a Single-Valued Reference to a Set Element. In the $Lift_{existing}$; the current floor of the lift is defined with variable *cur_floor*, which is an element of set ($groundf. .topf$). This concept is defined in the DSL with reference liftPosition whose source and target classes have been already mapped with the rules above. The source has been mapped to a machine and the target to a set. To map *cur_floor* to reference liftPosition, we introduce the following invariant:

Listing 3: Single-valued EReference to a Set Element

INVARIANT
  $linked\_floors(liftPosition(linked\_lift)) = cur\_floor$

3.3.4 Mapping a Multi-Valued Reference to a Set. The variable *call_buttons* of machine $Lift_{existing}$ is the set of pressed buttons that allows one to select the floors to which the lift is to be moved. This concept is represented in the DSL via reference selectedFloors. This leads to the following invariant:

Listing 4: Multi-valued EReference to a Set

INVARIANT
  $linked\_floors[selectedFloors[\{linked\_lift\}]] = call\_buttons$

3.3.5 Mapping an Enumeration to Boolean Values. In $\text{Lift}_{\text{existing}}$ the Boolean variable *door_open* defines the status of the lift door (false if the door is closed and true if it is open). This concept is represented in the DSL via attribute Door of class Lift, whose type is enumeration DoorStatus. The same case happens for attribute Direction and variable *direction_up*. Mapping these concepts to each other introduces additional invariants to $\text{Lift}_{\text{linkage}}$:

Listing 5: EnumType to Boolean Values

```
INVARIANT
 (Door(linked_lift) = Closed ⇔ door_open = FALSE)
 ∧ (Door(linked_lift) = Open ⇔ door_open = TRUE)
 ∧ (Direction(linked_lift) = Down
     ⇔ direction_up = FALSE)
 ∧ (Direction(linked_lift) = Up ⇔ direction_up = TRUE)
```

3.4 Initialization

Once the data structures of both machines have been mapped to each other, using additional variables and invariants, we need to think about the initialization and the operations in order to keep the conformance of both models all along the animation. Machine $\text{Lift}_{\text{linkage}}$ includes machine $\text{Lift}_{\text{static}}$, which allows it to use the basic operations provided by the latter to update the state of its variables. $\text{Lift}_{\text{linkage}}$ includes or refines (depending on the strategy chosen by the user) machine $\text{Lift}_{\text{existing}}$. In both cases, the constants of $\text{Lift}_{\text{existing}}$ are visible and can be read by $\text{Lift}_{\text{linkage}}$. The initialization of the model is done as:

Listing 6: Initializing the Model

```
PROPERTIES
    card(groundf . . topf) = card(FLOOR)
INITIALISATION
    SetDirection(linked_lift,Up);
    SetDoor(linked_lift,Closed);
    UnsetSelectedFloors(linked_lift) ;
    SetLiftPosition(linked_lift, linked_floors⁻¹(groundf))
```

Constants *groundf* and *topf* are not assigned to values in the existing specification. However, the visual animation starts from a model built in the DSL tool and in which buildings and floors have been already created. Thus, the PROPERTIES clause indicates to ProB to choose any valuation of these constants that respects the existing number of objects defined in the DSL model. Regarding the INITIALISATION clause, in addition to the initialisation of the linkage variables (Cf. listings 1 and 2), it updates the DSL model such that it starts in a state that is conformant to the initialisation of machine $\text{Lift}_{\text{existing}}$ given in Figure 2. To this purpose we simply call basic operations provided by $\text{Lift}_{\text{static}}$ (*i.e.* `SetDirection`, `SetDoor`, `UnsetSelectedFloors` and `SetLiftPosition`). Figure 7 is a screenshot of Meeduse after the initialization.

The state view on the bottom left side of the figure shows that the linked_lift is LIFT1. Note that ProB gives the two possible initialisations of linked_lift (LIFT1 and LIFT2), and the user may select the one to be controlled with $\text{Lift}_{\text{existing}}$. In this case, we selected LIFT1. Besides, the DSL model deals with five floors, which are all mapped via linked_floors satisfying the underlying invariant. The liftPosition of LIFT1 is set to FLOOR1 because machine $\text{Lift}_{\text{existing}}$ starts with a lift that is at the ground floor.

3.5 Operations

We propose two strategies to build the dynamic semantics of the DSL by means of a linkage machine: (1) Refinement/Inclusion and (2) Inclusion/Inclusion. In both strategies, the static semantics machine is included in the linkage machine.

3.5.1 Refinement/Inclusion. In this strategy, the existing machine is refined by introducing the linkage variables and invariants, which suggests the refinement of the operations too. In Listing 7 below, we give the refinement of operation *move_up*. We copied, from $\text{Lift}_{\text{existing}}$, the body of the operation (from line `2.` to line `10.`) and we introduced a call to operation *setLiftPosition* (line `11.`). This basic setter of relation `position` is provided by $\text{Lift}_{\text{static}}$; its task is to change the position of the lift conforming to the linkage. The objective is to preserve the invariant of Listing 3, since only variable *cur_floor* is modified by operation *move_up*.

Listing 7: Refining Operation *move_up*

```
OPERATIONS
1.  move_up =
2.   SELECT
3.     door_open = FALSE
4.     ∧ cur_floor < topf
5.     ∧ direction_up = TRUE
6.     ∧ ∃ cc.((cc ∈ Z ) ∧ (cc > cur_floor) ∧ (cc ∈ call_buttons))
8.     ∧ (cur_floor ∉ call_buttons)
9.   THEN
10.    cur_floor := ((cur_floor)+(1)) ;
11.  SetLiftPosition(linked_lift,linked_floors⁻¹(cur_floor))
12. END
```

3.5.2 Inclusion/Inclusion. In this strategy the existing machine is included in the linkage machine so that its operations can be used without a need to copy their body like in the Refinement/Inclusion strategy. The idea is to synchronise the states of both machines during the animation. For every operation of the existing machine, we create a synchronisation operation that manages the evolution of the two machines. For example, operation *move_up_inc* below calls both *move_up* and *SetLiftPosition*.

Listing 8: Synchronizing Models

```
OPERATIONS
move_up_inc =
 BEGIN
  move_up ;
  SetLiftPosition(linked_lift, linked_floors ⁻¹(cur_floor))
 END
```

Both strategies have their advantages. The Inclusion/Inclusion provides a lightweight approach to the usage of operations. It is practical when the DSL or the existing machine are not yet finished and may be changed. The Refinement/Inclusion is better if one would like to continue the refinement process such that the variables from the existing machine are completely redefined using variables of the static semantics machine. This technique may be

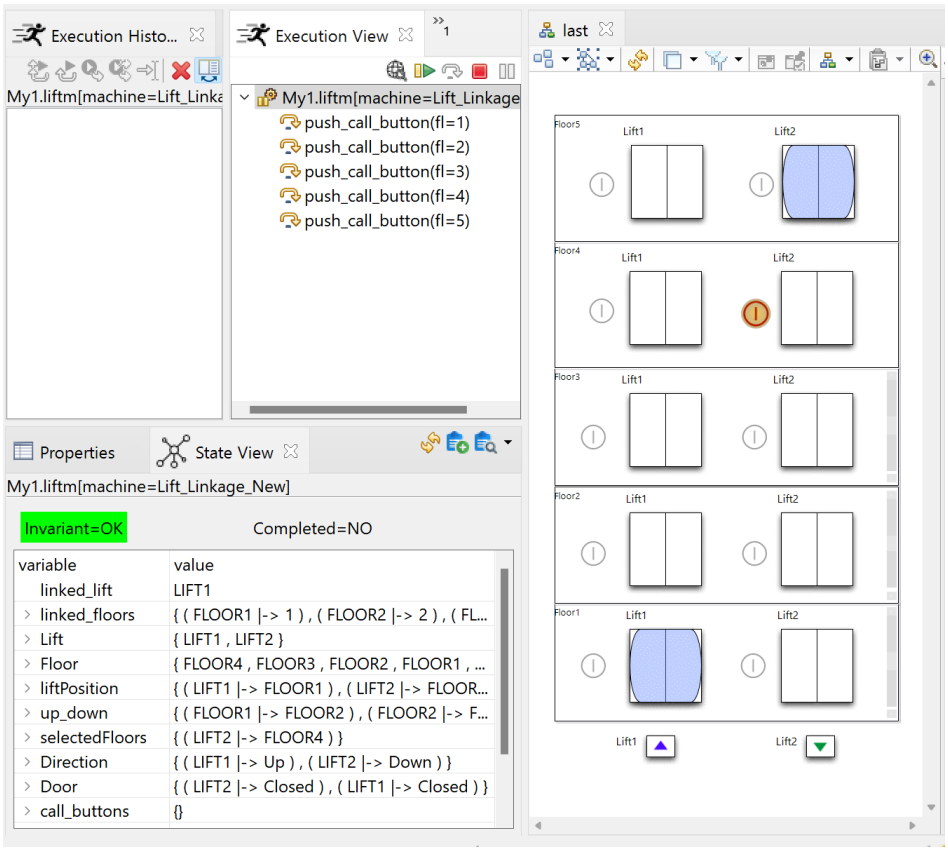

Figure 7: Meeduse after the initialization

useful to produce step-by-step the dynamic semantics of a DSL from an existing machine that has been already proved correct. In this paper, we do not go further in this approach since our objective is to focus on how a DSL can ensure visual animation of formal B specifications.

3.6 Enhancements

The initialization provided in listing 6 initializes $\text{Lift}_{\text{static}}$ based on the initial state of $\text{Lift}_{\text{existing}}$. This approach is reasonable as far as the objective is visual animation – since we do not introduce any modification to the existing specification, we just visualize it. Nevertheless, the limitation is that every time we switch from one lift to another (*e.g.* from LIFT1 to LIFT2) we must animate the initialization with a different lift. The selected new lift is, therefore, re-initialized, which brings it back to the ground floor. A better solution would be to use $\text{Lift}_{\text{existing}}$ as a controller which can switch between the two lifts. To this purpose, we need to initialize $\text{Lift}_{\text{existing}}$ from $\text{Lift}_{\text{static}}$ and introduce an operation that allows one to select on-the-fly any lift without re-initializing it. The setter (setValues) of listing 9 is introduced within $\text{Lift}_{\text{existing}}$ in order to update its variables conforming to its invariants. This kind of setter can be generated automatically: every parameter is assigned to a variable and the precondition applies the invariants to the parameters.

Listing 9: Adding a Setter to $\text{B}_{\text{existing}}$

```
OPERATIONS
setValues(call_buttons_,cur_floor_,direction_up_,door_open_) ==
 PRE
  cur_floor_ ∈ groundf..topf
   ∧ door_open_ ∈ BOOL
   ∧ call_buttons_ ∈ Pow (groundf . . topf)
   ∧ direction_up_ ∈ BOOL
   ∧ (door_open_ = TRUE ⇒ cur_floor_ ∈ call_buttons_)
 THEN
  cur_floor := cur_floor_
   || door_open := door_open_
   || call_buttons := call_buttons_
   || direction_up := direction_up_
 END
```

To initialize $\text{Lift}_{\text{existing}}$ from $\text{Lift}_{\text{static}}$ we define the substitution of listing 10 and we use it in the initialization clause of $\text{Lift}_{\text{linkage}}$. In this case, we are applying the Inclusion/Inclusion approach. Definition update_existing calls setter setValues based on the linkage invariants. Once the lift and the floors are mapped, this definition finds a valuation to the variables of $\text{Lift}_{\text{existing}}$ making them compatible with the valuations of $\text{Lift}_{\text{static}}$.

Listing 10: Initialize $B_{existing}$ from B_{static}

```
DEFINITIONS
update_existing ==
 ANY cur_floor_, door_open_, direction_up_, call_buttons_
 WHERE
  cur_floor_ = linked_floors(liftPosition(linked_lift))
   ∧ (Door(linked_lift) = Closed ⇒ door_open_ = FALSE)
   ∧ (Door(linked_lift) = Open ⇒ door_open_ = TRUE)
   ∧ (Direction(linked_lift) = Down
        ⇒ direction_up_ = FALSE)
   ∧ (Direction(linked_lift) = Up
        ⇒ direction_up_ = TRUE)
   ∧ linked_floors[selectedFloors[{linked_lift}]]
        = call_buttons_
 THEN
  setValues
  (call_buttons_, cur_floor_,direction_up_,door_open_)
 END
```

Finally, in order to switch between lifts at any time during the animation, we introduce the following operation select_lift. It selects a different lift, updates variable *linked_lift* and sequentially applies substitution update_existing. An illustration of this operation is provided in Figure 4. By running operation select_lift(LIFT2) one can control LIFT2 without bringing it back to the ground floor, and later continue with LIFT1. Note that operation select_lift could apply a scheduling approach and provide an optimized technique when switching between lifts.

Listing 11: Change Mapping On-The-Fly

```
OPERATIONS
select_lift =
    ANY lift WHERE
        lift ∈ Lift ∧ lift ≠ linked_lift
    THEN
        linked_lift := lift ; update_existing
    END
```

4 APPLICATION TO OTHER EXAMPLES

We applied our approach to several other examples, such as the *Scheduler* example discussed in [10, 12] and the ERTMS/ETCS railway system, which is a more realistic case study [14]. As the objective is to visualize a formal specification using a DSL in order to understand it, we do not present in this section the formal model. We rather provide screen-shots and identify some interesting notions that one can exhibit without looking at the formal specification.

Figure 8 applies state/transition diagrams to visualize the behaviour of processes managed by the Scheduler example. This model introduces three processes (one state/transition diagram per process), and deals with three states: Waiting, Ready and Active. Transitions refer to the various operations of the formal specification, which are: ready, active and swap. The highlighted transition shows the next enabled operations and the highlighted state shows the current state of the process.

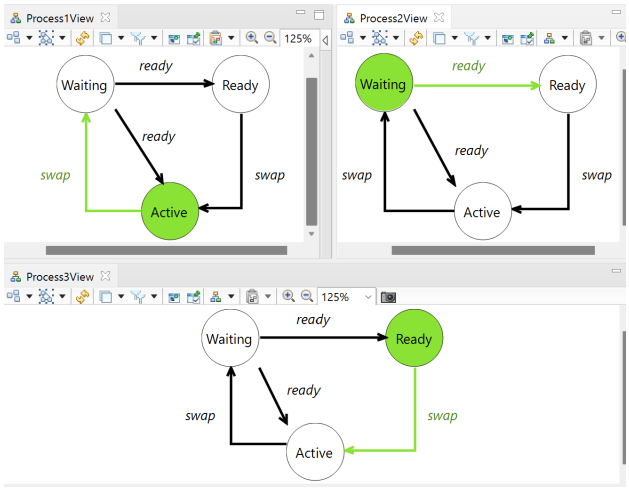

Figure 8: Graphical animation of the Scheduler example

Process1 showed in *Process1View* is in state Active whereas the animatable operation from this state is swap. *Process2View* and *Process3View* illustrate Process2 and Process3 whose current states are respectively Waiting and Ready. These state/transition diagrams emphasize on some properties such as deadlock freedom, or the non-blocking property. Indeed, one can see that after being active a process does not stop the system. In fact, the animation of operation swap puts Process1 in state waiting and activates one ready process. In this case, it activates Process3. As for the Lift, all the B specifications of the Scheduler can be found in the GIT repository[4].

Using Meeduse, we generated the Static B specification of the scheduler DSL. The generated specification includes three Sets and six variables (alongside their invariants and empty initialization), and operations (setters, getters, etc). This scheduler example helped us to explore a new mapping rule. In existing scheduler B specification, we have 3 variables: active, ready, waiting which are defined as sets of processes. We defined these concepts in the DSL via attribute Status, whose type is enumeration StatusEnum. Mapping enumerations to sets introduces the following invariants in the linkage machine:

Listing 12: EnumType to a Set

```
INVARIANT
  linked_process[dom(Status ▷ {Waiting}) ] = waiting
  ∧ linked_process[dom(Status ▷ {Ready}) ] = ready
  ∧ linked_process[dom(Status ▷ {Active}) ] = active
```

Lift and Scheduler are used as proof of concepts, which shows the applicability of our approach. We also applied the approach to a realistic railway system based on the formal B specification of [14]. The latter is composed of 4 components (1 machine and 3 refinements). This allowed us to evaluate the feasibility and scalability of our technique and to explore various other mapping rules. Figure 9 shows the underlying graphical animation. The current position of a train (called occupation) and the known position (called location) are illustrated via a table. The concept of TTD refers to a railway

[4]https://github.com/meeduse/Samples/tree/main/Scheduler

section. For example, Train1 occupies four sections (the rear is on TTD1 and the front is on TTD4), but the known position of the head of Train1 is TTD3. The right hand side of the figure shows the state of each TTD; it can be either occupied or free. From this view one can understand that the specification do not (yet) forbid accidents, since sections can be occupied by several trains.

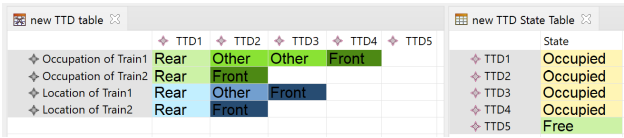

Figure 9: Graphical animation of the ERTMS/ETCS Case Study

The sizes of linkage specifications produced during the applicability of our approach are compared in Table 1. The comparison is based on produced linkage constants, properties, variables and invariants. The linkage constants are produced based on linking the constants and sets. Linkage properties are produced to define these constants or to refine existing constants. We only produced linkage variables during the lift example, in other examples the mapping of existing and DSL variables are done through linkage invariants. Table 1 also shows that the ERTMS example is 2 to 10 times bigger (in terms of lines) than the small examples of Lift and Scheduler mentioned in this paper.

Table 1: Comparison of Linkage Specifications

	Lift	Scheduler	ERTMS/ETCS			
			M0	M1	M2	M3
Linkage Constants		1	5	5	6	7
Linkage Properties	2	2	6	6	9	10
Linkage Variables	2					
Linkage Invariants	9	3	6	8	8	11
Size (lines) of Linkage Specification	117	105	247	341	415	1000

5 RELATED WORK AND DISCUSSION

Mixing formal methods and domain-specific languages has been addressed in several works [2, 15, 17, 23], but often with the intention to provide formal semantics to a DSL and hence be able to reason about its correctness. In fact, the strength of formal methods originates from their precision and the availability of automated reasoning tools. An exhaustive state of the art about this topic is provided in [5]. The work presented in this paper addresses a reverse approach: use a DSL as a way to remedy the poor readability of a formal specification, since the strength of DSLs is the domain-centric notations. Zalila *et al.,* in [23] suggest the use of an additional DSL to feedback formal verification results. The authors propose a new language (called FeVeRel) that allows the designer to implement the verification result feedback from the formal level to the DSL level. The approach is close to visual animation, but the use of additional languages may weaken the acceptance of the underlying technique because it requires new skills. The work of Tikhonova [20] starts from a DSL meta-model, generates Event-B specifications, but applies classical visual animation using BMotion Studio to the formal specifications in order to understand the behavior of the DSL. The limitation is that the DSL syntax must be reworked in BMotion Studio, which requires additional verifications to check the compatibility between the initial DSL syntax and the graphical visualization.

Our proposal is to use the DSL itself in order to ensure visual animation. As far as we know this technique has not been investigated before, mainly because, apart from Meeduse, none of the existing language workbenches provide execution features that are built on a well-established formal method such as B. This claim is attested by the survey of Iung *et al.,* [6] in which the formal dimension is completely absent. This survey shows that among the 59 discussed language workbenches only 9 provide support for verification, which is only done by testing. The interesting aspect of Meeduse in the current work is that it builds on the B method and applies ProB to execute the DSL. The approach does not need any additional implementation effort. Once the DSL is done, one can re-use a formal specification that is provided by a formal methods expert to execute it.

It is known that visual animation is highly desirable when applying formal methods tools because the poor readability of formal notations prevents user involvement during the validation activity. The approach of this paper complements the numerous tools and efforts dedicated to the B method: BRAMA [18], AnimB[5], B-Motion Studio [8], B-Motion Web [9], and VisB [22]. The added value with regards to these tools is that the B method expert only applies B to link his/her specifications to the formal static semantics of the DSL. In [12], Leuschel *et al.,* propose a simple way of producing custom animations. The idea is to assemble a series of pictures and write an animation function in B itself, which stipulates which pictures should be shown where depending on the current state of the system. Our approach builds on a similar argument but starts from graphical models that are designed by domain experts using a DSL tool crafted by MDE expert.

6 CONCLUSION

This paper presented an approach for the visual animation of B specifications, using an MDE paradigm built on DSLs. Our objective was to ensure validation thanks to domain-centric notations that are expected to be more comprehensible for domain experts than the formal specifications. We used Meeduse, a language workbench dedicated to formally instrument the static and dynamic semantics of a DSL by means of B models.

We provided a linkage B specification which maps the B models managed by Meeduse and the B specification to visualize. Two strategies are provided and discussed in this paper: (*i*) Refinement/Inclusion, and (*ii*) Inclusion/Inclusion. The examples mentioned in this paper are covered to show the viability of our technique. Writing the linkage B specification is easier when the DSL is based on the same concepts as the existing B specification. DSLs can be also used to visualize specifications that are defined with

[5]AnimB: https://wiki.event-b.org/index.php/AnimB

several refinement levels. For space reason, we did not develop this aspect here.

Besides, we showed that our approach also favors the reuse of existing specifications as dynamic semantics of a DSL. It can be done with minor modifications of the existing specification (*e.g.* by adding operations such as setValues and update_existing). During this work, the linkage B specifications were written manually but systematically. Future work will exploit the systematic character to generate linkage specifications (semi)-automatically. We intend to design an additional DSL to express the linkage. It will be integrated in Meeduse and will simplify the work of B method expert.

REFERENCES

[1] J.-R. Abrial. 1996. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, USA.

[2] Jean-Paul Bodeveix, Mamoun Filali, Julia Lawall, and Gilles Muller. 2005. Formal Methods Meet Domain Specific Languages. In *Proceedings of the 5th International Conference on Integrated Formal Methods*. Springer, 187–206. https://doi.org/10.1007/11589976_12

[3] Eclipse Foundation 2022. Eclipse Sirius. Retrieved September 5, 2022 from https://www.eclipse.org/sirius/

[4] Akram Idani. 2020. Meeduse: A Tool to Build and Run Proved DSLs. In *Integrated Formal Methods*, Brijesh Dongol and Elena Troubitsyna (Eds.). Springer International Publishing, Cham, 349–367.

[5] Akram Idani, Yves Ledru, and Germán Vega. 2020. Alliance of model-driven engineering with a proof-based formal approach. *Innov. Syst. Softw. Eng.* 16, 3 (2020), 289–307. https://doi.org/10.1007/s11334-020-00366-3

[6] Aníbal Iung, João Carbonell, Luciano Marchezan, Elder Macedo Rodrigues, Maicon Bernardino, Fabio Paulo Basso, and Bruno Medeiros. 2020. Systematic mapping study on domain-specific language development tools. *Empirical Software Engineering* 25, 5 (2020), 4205–4249.

[7] Sebastian Krings and Philipp Körner. 2021. Prototyping Games Using Formal Methods. In *Formal Methods – Fun for Everybody*, Antonio Cerone and Markus Roggenbach (Eds.). Springer International Publishing, Cham, 124–142.

[8] Lukas Ladenberger, Jens Bendisposto, and Michael Leuschel. 2009. Visualising Event-B Models with B-Motion Studio. In *Formal Methods for Industrial Critical Systems*, María Alpuente, Byron Cook, and Christophe Joubert (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 202–204.

[9] Lukas Ladenberger and Michael Leuschel. 2016. BMotionWeb: A Tool for Rapid Creation of Formal Prototypes. In *Software Engineering and Formal Methods*, Rocco De Nicola and Eva Kühn (Eds.). Springer International Publishing, Cham, 403–417.

[10] Bruno Legeard, Fabien Peureux, and Mark Utting. 2002. Automated Boundary Testing from Z and B. In *FME 2002:Formal Methods—Getting IT Right*, Lars-Henrik Eriksson and Peter Alexander Lindsay (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 21–40.

[11] Michael Leuschel and Michael Butler. 2003. ProB: A Model Checker for B. In *FME 2003: Formal Methods*, Keijiro Araki, Stefania Gnesi, and Dino Mandrioli (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 855–874.

[12] Michael Leuschel, Mireille Samia, and Jens Bendisposto. 2008. Easy graphical animation and formula visualisation for teaching B. (2008).

[13] Shaoying Liu and Weikai Miao. 2021. A formal specification animation method for operation validation. *Journal of Systems and Software* 178 (2021), 110948.

[14] Amel Mammar, Marc Frappier, Steve Jeffrey Tueno Fotso, and Régine Laleau. 2020. A formal refinement-based analysis of the hybrid ERTMS/ETCS level 3 standard. *International Journal on Software Tools for Technology Transfer* 22, 3 (June 2020), 333–347. https://doi.org/10.1007/s10009-019-00543-1

[15] Bart Meyers, Hans Vangheluwe, Joachim Denil, and Rick Salay. 2020. A Framework for Temporal Verification Support in Domain-Specific Modelling. *IEEE Transactions on Software Engineering* 46, 4 (2020), 362–404. https://doi.org/10.1109/TSE.2018.2859946

[16] Monika Müllerburg, Leszek Holenderski, Olivier Maffeïs, Agathe Merceron, and Matthew Morley. 1995. Systematic testing and formal verification to validate reactive programs. *Softw. Qual. J.* 4, 4 (1995), 287–307. https://doi.org/10.1007/BF00402649

[17] José Rivera, Francisco Durán, and Antonio Vallecillo. 2009. Formal Specification and Analysis of Domain Specific Models Using Maude. *Simulation* 85 (10 2009), 778–792.

[18] Thierry Servat. 2006. BRAMA: A New Graphic Animation Tool for B Models. In *B 2007: Formal Specification and Development in B*, Jacques Julliand and Olga Kouchnarenko (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 274–276.

[19] Dave Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. 2009. *EMF: Eclipse Modeling Frameworke* (2nd ed.). Addison-Wesley.

[20] U. Tikhonova, M.W. Manders, M.G.J. Brand, van den, S. Andova, and T. Verhoeff. 2013. Applying model transformation and Event-B for specifying an industrial DSL. In *Workshop on Model Driven Engineering, Verification and Validation (CEUR Workshop Proceedings)*. CEUR-WS.org, 41–50.

[21] Nathaniel Watson, Steve Reeves, and Paolo Masci. 2018. Integrating User Design and Formal Models within PVSio-Web. *Electronic Proceedings in Theoretical Computer Science* 284 (11 2018). https://doi.org/10.4204/EPTCS.284.8

[22] Michelle Werth and Michael Leuschel. 2020. VisB: A Lightweight Tool to Visualize Formal Models with SVG Graphics. In *Rigorous State-Based Methods*, Alexander Raschke, Dominique Méry, and Frank Houdek (Eds.). Springer International Publishing, Cham, 260–265.

[23] Faiez Zalila, Xavier Crégut, and Marc Pantel. 2016. A DSL to Feedback Formal Verification Results. In *13th Model-Driven Engineering, Verification and Validation Workshop at MODELS conference 2016 (MoDeVVa 2016)*, Michalis Famelis, Daniel Ratiu, and Gehan M. K. Selim (Eds.), Vol. 1713. CEUR-WS : Workshop proceedings, Saint Malo, France, 30–39. https://hal.archives-ouvertes.fr/hal-03172263