Innovations in Systems and Software Engineering
https://doi.org/10.1007/s11334-020-00366-3

ORIGINAL ARTICLE

Alliance of model-driven engineering with a proof-based formal approach

Akram Idani[1] · Yves Ledru[1] · German Vega[1]

Received: 30 November 2018 / Accepted: 18 May 2020

Abstract
Model-driven engineering (MDE) promotes the use of models throughout the software development cycle in order to increase abstraction and reduce software complexity. It favors the definition of domain-specific modeling languages (DSMLs) thanks to frameworks dedicated to meta-modeling and code generation like EMF (Eclipse Modeling Framework). The standard semantics of meta-models allows interoperability between tools such as language analysers (e.g., XText), code generators (e.g., Acceleo), and also model transformation tools (e.g., ATL). However, a major limitation of MDE is the lack of formal reasoning tools allowing to ensure the correctness of models. Indeed, most of the verification activities offered by MDE tools are based on the verification of OCL constraints on instances of meta-models. However, these constraints mainly deal with structural properties of the model and often miss out its behavioral semantics. In this work, we propose to bridge the gap between MDE and the rigorous world of formal methods in order to guarantee the correctness of both structural and behavioral properties of the model. Our approach translates EMF meta-models into an equivalent formal B specification and then injects models into this specification. The equivalence between the resulting B specification and the original EMF model is kept by proven design steps leading to a rigorous MDE technique. The AtelierB prover is used to guarantee the correctness of the model's behavior with respect to its invariant properties, and the ProB model-checker is used to animate underlying execution scenarios which are translated back to the initial EMF model. Besides the use of these automatic reasoning tools in MDE, proved B refinements are also investigated in this paper in order to gradually translate abstract EMF models to concrete models which can then be automatically compiled into a programming language.

Keywords Software systems · Model-driven engineering · Formal methods · Visual animation · Proofs

1 Introduction

Model-driven engineering (MDE) is a promising paradigm in software systems development because it provides solutions to the software complexity on the one hand, and it shows how to bridge the gap between conceptual models and coding activities, on the other hand. The definition and the use of domain-specific models throughout the engineering life-cycle makes MDE a powerfull asset. Furthermore, MDE is assisted by numerous tools (EMF,[1] XText,[2] ATL,[3] etc) dedicated to put into practice a clear separation of concerns ranging from requirements to target platforms, and going through several design stages. Interoperability between these tools is favored by the use of standardized meta-modeling formalisms which increases automation especially for developing domain specific modeling languages (called DSMLs). These advantages allow to reduce the risk that human errors such as misinterpretation of the requirements and specification documents lead to erroneously validate the specification, and hence to produce the wrong system. Still, while MDE provides solutions to the validation problem ("do the right system"), the verification problem ("do the system right") remains a major challenge. Indeed, most of the verification activities offered by MDE tools are based on the verification of OCL constraints given instances of meta-models. However, these constraints mainly deal with properties of the structure of the model, but not of its behavior. There is hence noway in MDE to prove invariant preservation while execut-

[1] EMF: https://www.eclipse.org/modeling/emf/.
[2] Xtext: https://www.eclipse.org/Xtext/.
[3] ATL: http://www.eclipse.org/atl/.

✉ Akram Idani
akram.idani@imag.fr

[1] Univ. Grenoble Alpes, Grenoble INP, CNRS, LIG, 38000 Grenoble, France

Published online: 01 June 2020

ing a DSL operational semantics. This limitation is due to the lack of formal reasoning tools that allows to ensure the correctness of models.

This paper proposes a new method to define mappings between DS(M)Ls as they are defined in MDE techniques and the rigorous world of formal methods. Our approach is assisted by Meeduse,[4] a tool that we developed in order to translate a DS(M)L meta-model into an equivalent formal B specification that represents its semantic domain, using the set theory and the first-order predicate logic. The proposed method consists in translating the DS(M)L data structures and multiplicity constraints to abstract data types (sets, variables, etc) and primitive functions for the creation, destruction, reading and updating of data, such that all the meta-model structural constraints are taken into account by means of pre- and post-conditions. The resulting formal specification guarantees by construction the preservation of the data structure properties when high-level functions are used to define the DS(M)L behavior.

In MDE, the design of a DS(M)L addresses two main layers: meta-modeling and modeling. Meta-modeling refers to the definition of the language abstract syntax where domain concepts and their relationships are defined independently from their concrete representation. The modeling layer allows to create instances of these domain concepts by applying a predefined concrete syntax (graphical or textual). This paper discusses how the formal B language is applied to both layers based on a simple running example. At the modeling layer, the abstract data-types generated from the meta-model layer, become valuated data types which allows animations of primitive functions sequences on given models. Regarding the DSL behavioral semantics, they follow the same principles. Indeed, they are defined over the meta-modeling layer using high-level B operations, which are then animated by Meeduse on the modeling layer. This alliance between MDE and a formal method, assisted by Meeduse, makes domain-specific models provable and also executable thanks to the animation of their expected behavior directly in a dedicated DS(M)L tool. The interest of Meeduse is that it integrates the ProB [26] animator. Given a model designed in the DS(M)L tool, Meeduse injects it as valuations in the generated B specification and calls ProB in order to compute the list of operations that may be animated from these valuations. When an operation is animated, the tool computes the new variable valuations and then it translates back these valuations to the initial model which results in an automatic animation of domain models.

Our approach allows to take benefit of formal reasoning tools such as provers and model-checkers throughout the MDE process. The proposed method uses the AtelierB prover to guarantee the correctness of the model's behavior with respect to its invariant properties, and the ProB model-checker in order to animate underlying execution scenarios. Besides the use of these automatic reasoning tools in MDE, proved B refinements are investigated in order to gradually translate abstract models to concrete ones which can then be automatically compiled into a programming language.

Section 2 defines the meaning of DS(M)Ls semantics and presents how our approach is useful in order to formally define these semantics. The next sections of the paper show through a practical example, how proofs and refinements can be applied during the definition of a DS(M)L by using the Meeduse tool: at the meta-modeling level (Sect. 3), the modeling level (Sect. 4) and when specifying model execution (Sect. 5). Then, Sect. 6 provides a state of art about existing works that tried to define formal DS(M)L semantics and compare them to our approach. In Sect. 7, we describe two real-life applications of our approach and we discuss their results. Finally, Sect. 8 draws the conclusions and the perspectives of this work.

2 DS(M)L's semantics

Execution of domain-specific (modeling) languages (DS(M)Ls) is an active research area in model-driven engineering (MDE). The intention is to be able to perform early analysis of a system's behavior before its implementation. Indeed, a DS(M)L execution tends to reduce the gap between the model and the system, since DS(M)Ls would not only represent the expected system's structure and behavior but they can themselves behave as the system should run. In this work, we apply a formal approach in order to define, prove, and execute the underlying semantics of a DS(M)L. This section presents an overview about DS(M)L semantics as they are defined in the state of the art and explains the main principles of our approach.

2.1 Abstract syntax and semantic domains

In [4], Bryant et al. discuss challenges and directions about DSLs semantics and point out that their formal definition is the foundation for several benefits: automatic generation of some DSL tooling, formal analysis of a model's behavior, and/or composition of concerns issued from different languages. The authors define the semantics of a DSL as a mapping from its abstract syntax to some semantic domain. The abstract syntax defines the modeling concepts, their relationships and attributes, and the semantic domain is some mathematical framework whose meaning is well-defined. In the same direction, Harel et al. [13] state that a semantic domain is a well-defined and well-understood agreement on a language's meaning.

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

In MDE, the notion of abstract syntax is well mastered today thanks to the concept of meta-model (which is standardized by the Meta-Object Facility MOF [12] document). However, the notion of semantic domain remains unclear because, on the one hand, the MOF is informally defined, and on the other hand, a common misconception in modeling languages is to confuse semantics with behaviors [13]. In fact, the description of a behavior is itself a DSL with its own abstract syntax and semantic domain. For example, several MDE tools [3,7,14,23,28,43] define the simple Petri-net DSL using a common meta-model, but they apply various languages (Kermeta, xMof, fUML, java) to define the expected behavior when firing a Petri-net transition. These languages are additional DSLs with different semantic domains that lead to different behaviors when executing the same Petri-net model with different tools.

Meta-models are then not sufficient to describe the behavior of a DSL, that's why some attempts exist in order to enhance meta-models by action languages (e.g., Kermeta [3]), or to translate meta-models into a target language that offers behavioral facilities (e.g., Java [14]). In our approach, we apply this second strategy (called translational in [4]) but we use the formal B language [1] which offers a way to mathematically define the semantic domain of a DSL. Terms "shallow embedding" and "deep embedding" [44] are often used to describe a mapping between formalisms. The first notion means a direct translation from a source model into a target model, while the second notion means that the mapping leads to structures that represent data types. In our approach, we adopt a deep embedding approach when translating a DSL meta-model into the B language. The formal semantic domain of a DSL will represent data typing with the corresponding structural constraints defined in set theory and first-order logic. Regarding the DSL behavior, it is defined using B operations which can be manually written, like presented in this paper or generated from additional DSLs that can describe behaviors. In our previous works [17], we proposed to produce the B operations from a coloured Petri-nets DSL. Nonetheless, in this paper, in order to avoid the use of several formalisms, we choose to manually write these B operations.

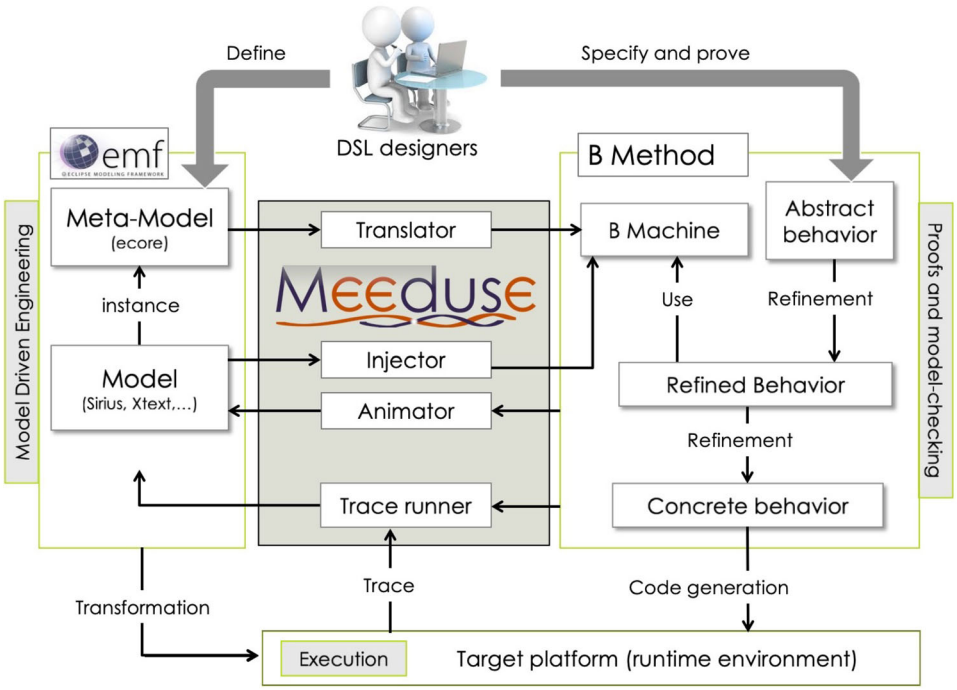
2.2 Dynamic semantics versus behavioral semantics

Several research works have been devoted in order to provide solutions with tools that make DSLs executable. In model-driven engineering, DSL execution mainly refers to the definition of semantics, which are roughly called by means of several terms like dynamic, operational, executable and/or behavioral. In [5] for example, it is mentioned that DSL execution gives a "*dynamic facility to observe and control an* execution *in order to better understand a* behavior *or to look for the cause of a defect*". In [20], the authors refer to behavioral semantics of a DSL, as the semantics that allow run a program. They identify two kinds of behavioral semantics: translational semantics and operational semantics: A translational semantics would result in a compiler while an operational semantics would result in an interpreter. In [30], a different definition is provided without a distinction between operational semantics and behavioral semantics. The authors refer to a high level execution defined over meta-models and which uses actions (also called operations) to define model's behavior. Similarly to programs which are made of data structures and algorithms, in [30] an executable DSL is made of meta-data and actions.

We think that the terminology around semantics of DSLs that makes them executable is not well established in the MDE literature and they are somehow confusing. In fact, the various definitions that we found mix low-level notions (compilers, programs, etc), and high level notions (meta-models, etc). This is less the case for programming languages where the terms dynamic semantics and execution semantics are used to describe the runtime behavior of programs [42]. The so-called dynamic (or execution) semantics in programming languages, "*defines how and when the various constructs of the language produce artifacts* (e.g., bytecode) *that can be executed in a given target platform* [10] (e.g., virtual machine)".

For clarity we use terms dynamic and execution semantics of a DSL as soon as a runtime environment is concerned which is close to programming languages [10,42]. Terms operational and behavioral semantics for us are inspired by [30] and they are limited to high level actions that describe how models behave. The gap between both semantics does not make them conflicting because at a high level one can define behaviors of models and at a low level this behavior can be translated into a runtime environment. The challenge is then to guarantee the compatibility between the resulting executions: that of a model given its operational semantics, and that of the runtime program given its dynamic semantics. Works of Tikhonova [39–41] bridges this gap between operational and dynamic semantics, by providing reusable specification templates which are templates that allow to explicitly define the dynamic semantics of a DSL over its operational semantics. These templates are presented in form of a library of specifications, each of which formalizes a separate software solution. The operational semantics of a DSL are then expressed as a composition of these specification templates, which favors some kind of generic programming of the DSL dynamic semantics. However, the use of a specification templates library makes several restrictions to the definition of semantics because it is not obvious to cover all possible behavioral needs with various abstraction levels. In our proposal, we address the semantic gap between operational and dynamic semantics of a DSL thanks to the refinement paradigm of the B method which allows to pro-

Fig. 1 Meeduse overview



duce step-by-step a low level implementation of a DSL from its high level description.

2.3 Executable DS(M)Ls in Meeduse

Figure 1 summarizes how we put into practice the alliance of MDE with a proof-based formal approach in order to define DS(M)Ls with formal semantics and favor their execution. Our tool support named Meeduse[5] links three technological spaces: EMF [38] for model-driven engineering, B Method [1] for proofs and refinements, and finally the execution of the target system. The top hand side of Fig. 1 represents the abstract syntax of a DSL by means of an ECore meta-model, its mapping to the corresponding formal semantics domain (box B Machine), and its operational semantics which starts from the description of an abstract behavior and applies refinements until the specification of its concrete behavior (boxes abstract behavior, refined behavior and concrete behavior). The bottom hand side of Fig. 1 defines the dynamic semantics of the DSL since it is an extraction of an executable language from the high level DSL artifacts.

We build our approach on several components: Translator, Injector, Animator and Trace runner.

(1) Translator: this component translates an Ecore [38] meta-model into an equivalent B specification which gathers the structure of the meta-model as well as basic operations like constructors, destructors, getters and setters. The resulting B specification can be enhanced manually by additional invariants and then its proof of correctness with respect to these invariants can be done using the AtelierB prover.

(2) Injector: injects instances of a meta-model, which can be designed using EMF-based modeling tools (like Sirius, GMF, XText, etc) into the B specification produced from the meta-model. This component introduces enumerations into abstract data structures like abstract sets, and hence allows valuations of the B machine variables.

(3) Animator: in Meeduse, animation of B specifications is done using the ProB tool [26] which is an open-source model-checker supporting the B method. The Animator component asks ProB to animate B operations and gets the reached state by means of B variables valuations. Then, Meeduse computes the equivalence between these valuations and the initial EMF model and applies the necessary modifications to the model in order to maintain this equivalence all along the animation process.

(4) Trace runner: this component allows to play a sequence of operations issued from an execution trace by animating the corresponding B operations which leads to automatic modifications of the model. Thanks to this component, animation can be done from outside EMF by an external program which is running in the target platform. Note that the target platform source code can be produced by applying MDE transformations or by using the code generator of AtelierB after successive refinements. In both cases, the trace runner is useful on the

[5] Meeduse: Modeling Efficiently EnD USEr needs (http://vasco.imag.fr/tools/meeduse/).

one hand, for conformance validation between the model and the execution, and on the other hand for some kind of runtime verification. In this paper, we only present the first strategy where an implementation in the C language is generated by AtelierB from the most concrete B machine. As the C generator of AtelierB is certified, we have the guarantee that the resulting implementation preserves the model's properties.

In our approach, the B specifications (right-hand side of Fig. 1) are structured in several parts which are presented in the following sections: formal semantic domain (called B Machine in Fig. 1 and presented in Sects. 3, 4), operational semantics (represented by the abstract behavior, refined behavior and concrete behavior, introduced in Sect. 5).

3 Meta-modeling

The meta-model notion is central in MDE because, on the one hand, it defines model structure and its underlying abstract syntax, and on the other hand, it allows interoperability between tools such as language analysers (e.g., XText [2]), code generators (e.g., Acceleo [9]), and also model transformation tools (e.g., ATL [21]). Nonetheless, the relevance of a meta-model and associated tools depend on their capacity to capture domain specific concepts and also on how useful they are in modeling, discussing and agreeing on the domain-specific requirements from the end user point of view.

3.1 A simple example

Figure 2 gives a simple Ecore meta-model defined using the EMF tool in order to capture the domain-specific concepts of a parking lot. These concepts are:

1. The parking lot itself (designed by meta-class *Parking*) with the number of cars that stay in the parking (attribute *nbCars*),
2. The set of places (meta-class *Place*) which can be empty or not (Boolean attribute *empty*), and
3. The red/green light (meta-class *Light*) to forbid or authorize the access to the parking.

EMF generates automatically a standard Java implementation from a meta-model allowing to programmatically create and modify instances of this meta-model. It also produces an Eclipse plugin editor based on this Java implementation. The correctness of the resulting Java code is an open and crucial issue for a rigorous modeling since this Java code does not cover most of the implicit structural constraints. For example,

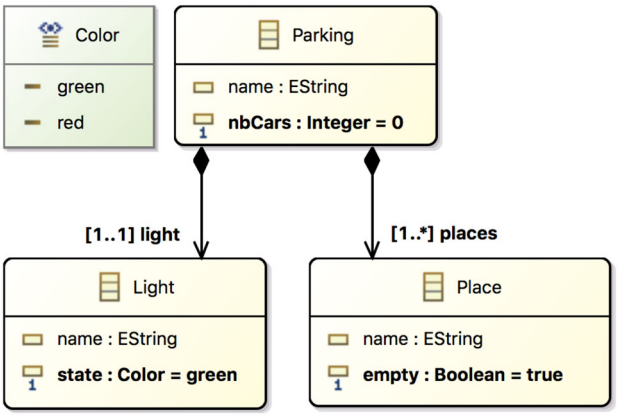


Fig. 2 Parking meta-model

```
/**
 * @generated
 */
public Parking createParking() {
    ParkingImpl parking = new ParkingImpl();
    return parking;
}
```

Fig. 3 Java code of parking creation in EMF

Fig. 3 provides the Java implementation produced by EMF in order to create instances of class Parking. This method uses the default constructor of class ParkingImpl. A call to this method creates then in an uncontrolled way a parking object leading then to an erroneous model with respect to the multiplicities. Indeed, as mentioned before, an instance of the parking class must be linked to one and only one light, and must have at least one place.

This limitation concerns several other operations generated by EMF. For example, operation $setLight$ which links a light to a parking, does not verify whether the new light is already linked to an other parking, etc. This permissiveness of EMF makes the resulting models subject to many errors. In order to address this limitation, we produce a formal specification of the meta-model and of the various operations allowing to manage the corresponding instances.

3.2 Extraction of B specifications

Figure 4 shows the B data structures with their corresponding typing invariants produced automatically by Meeduse from the meta-model of Fig. 2. This translation from Ecore meta-models into B specifications applies a sequence of transformation rules which can be summarized by:

– A meta-class $Class$ gives an abstract set named $CLASS$ representing possible instances and a variable named

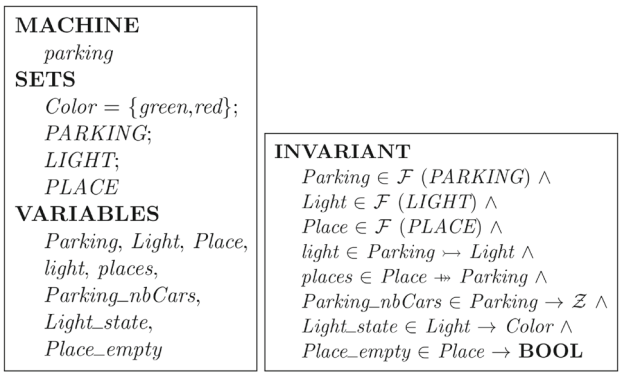


Fig. 4 B data structures produced from the meta-model

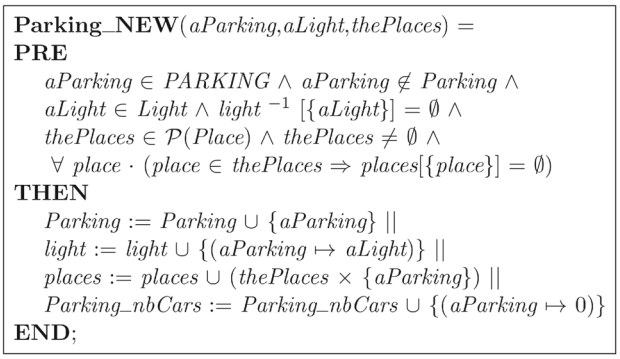


Fig. 5 Instantiation of class Parking in B

Class representing the set of existing instances such that existing instances belong to the set of possible instances. For example, meta-class Parking is specifyied by abstract set $PARKING$, variable $Parking$ and invariant $Parking \in \mathcal{F}(PARKING)$. Expresion $\mathcal{F}(PARKING)$ designs the set of finite subsets over $PARKING$.

- An enumeration is translated into a given set (e.g., Color).
- Basic types (e.g., integer, boolean) become B types (Z, Bool, etc).
- Attributes and references lead to functional relations depending on multiplicities and their optional/mandatory character. For example, containment reference light is translated into an injective total function from set $Parking$ to set $Light$.

The behavioral part of the generated B machine gathers all basic operations: getters, setters, constructors and destructors. These operations are correct by construction with respect to the invariants produced automatically from the meta-model structure. Figure 5 gives the B specification of operation Parking_NEW which creates instances of class Parking. This operation introduces an element called $aParking$ in set $Parking$ given several pre-conditions. First, it must have a light in the model ($aLight \in Light$) which is not already linked to a parking ($light^{-1}[\{aLight\}] = \emptyset$). Then, the set of places (called $thePlaces$) which will be associated to the parking should contain at least one place ($thePlaces \neq \emptyset$). And finally, these places must not belong to a parking ($\forall place \cdot (place \in thePlaces \Rightarrow places[\{place\}]) = \emptyset$). Provided these preconditions, actions of operation Parking_NEW create instance $aParking$ and link it, on the one hand, to the light specified by parameter $aLight$, and on the other hand to all places of set $thePlaces$. As mentioned in the meta-model, attribute $nbCars$ is initialized to 0.

Contrary to the Java implementation (Fig. 3) produced by EMF, this B specification instanciates safely class Parking regarding its structural requirements. The model designer must then create a light and at least one place, and must verify that these objects are not linked to an other parking, before being able to call operation Parking_NEW.

For this example, our approach produced 18 B operations (constructors, destructors, getters and setters) for which the AtelierB prover generated 48 proof obligations (POs) and was able to automatically prove 38 OPs. The 10 remaining POs were proved interactively without any improvement or modification of the B specifications. In fact, B specifications produced by Meeduse from the meta-model are correct by design with respect to the structural invariants.

3.3 Introducing constraints

Besides the implicit constraints defined by multiplicities, mandatory attributes and their initial values, MDE advocates for the use of OCL [31] in order to specify additional structural constraints. For our example, the following constraint means that the number of cars in the parking is equal to the number of non-empty places.

```
Context Parking inv nbCarsConstraint:
  self.nbCars =
  self.places->select(not empty)->size()
```

In EMF, there is no way to detect methods which may introduce violations of OCL constraints and then the EMF source code is even more error-prone. In order to take into account this constraint we formalize it directly in B by introducing the following invariant[6]:

$$\forall\, parking \,.\, (parking \in Parking \Rightarrow Parking\_nbCars(parking) = \mathbf{card}((places^{-1}[\{parking\}] \lhd Place\_empty) \rhd \{\mathbf{FALSE}\}))$$

[6] Symbols $\lhd$ and $\rhd$ represent, respectively, domain and range restrictions.

```
Place_SetEmpty(aPlace,aEmpty) =
  PRE
    aPlace ∈ Place ∧ aEmpty ∈ BOOL
    ∧ places[{aPlace}] ≠ ∅ ∧ aEmpty ≠ Place_empty(aPlace)
  THEN
    IF aEmpty = TRUE THEN
      Parking_nbCars(places(aPlace)) := Parking_nbCars(places(aPlace)) - 1
    ELSE
      Parking_nbCars(places(aPlace)) := Parking_nbCars(places(aPlace)) + 1
    END ||
    Place_empty(aPlace) := aEmpty
  END
```

Fig. 6 Improved B operation

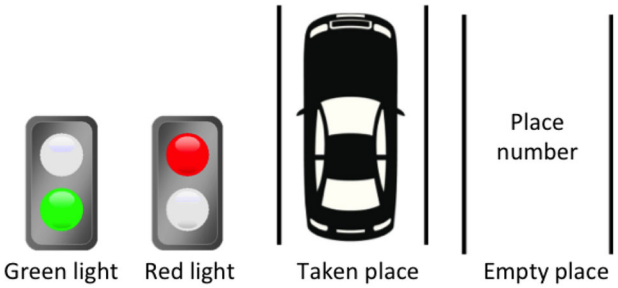


Fig. 7 Palette of graphical representations

Note that we did not define a translation from OCL into B, and hence constraints are manually written as invariants in the B specification. The automatic translation from OCL to B is an interesting perspective for our work, however we think that it may produce complex invariants which make the proof activity somehow more difficult. The above invariant led to 15 additional POs allowing to detect generated B operations which may violate this constraint. For example, operation Parking_NEW of Fig. 5 violates this invariant because it assigns value 0 to attribute $nbCars$ and because parameter thePlaces may include non-empty places. The failure of this proof shows an inconsistency between the meta-model and the constraint. Indeed, the meta-model assigns a default value (0 in this case) which is not conformant to the required value. The proof fails also for several other operations like:

- Parking_AddPlaces and Parking_RemovePlaces: when adding or removing a set of places, attribute nbCars must be updated depending on the number of new not empty places.
- Parking_SetNbCars: the setter of attribute nbCars must check the new value assigned to this attribute.
- Place_SetEmpty: when a place becomes empty or occupied, the number nbCars must be increased or decreased consequently.

Figure 6 gives the B specification of operation Place_SetEmpty after improvements.

We added clause IF...THEN...ELSE allowing to update correctly attribute nbCars and we also strenghened its precondition in order to be called only on places which are already linked to a parking ($places[\{aPlace\}] \neq \emptyset$). Regarding operation $Parking\_NEW$, two ways are possible to correct it: revise the last substitution of Fig. 5 such that the attribute assignment becomes conformant to the required value or add a precondition reducing parameter thePlaces to empty places ($Place\_Empty[thePlaces] = \{TRUE\}$). Finally, operation Parking_SetNbCars is removed since attribute nbCars should not be set explicitly by the user.

Discussion. The java constructor of Fig. 3 and other error prone modeling operations are generated by EMF from the meta-model of Fig. 2 and used by the corresponding model editor. Hence, as several existing tools are built on top of EMF (model execution, model transformation, etc), they remain also error prone. It is then difficult to detect from the resulting implementations the origin of errors or inconsistencies when the meta-model's invariant properties are violated. This section showed the interest of a proof-based meta-modeling in comparison with a classical technique were a Java source code is produced from a meta-model in order to provide a set of programmatic operations for model manipulation. Our approach addresses this challenge by making possible the use of a prover at a meta-modeling level. On the one hand, this forces the designer to identify and improve the critical parts of his modeling operations, and on the other hand it produces a set of operations which guarantee the construction of a correct model with respect to structural invariants as well as additional constraints. By applying Meeduse, the designer has two choices: play with EMF to create the model (in that case he must be sure that his model is correct at least structurally) or play with our B modeling operations (in that case, the model is kept correct all along the design actions). Thus, the main intention of our work is to define behavioral semantics that allow model execution. Existing works that define behavioral semantics basing on the EMF generated code lack of rigor because of the discussed EMF limitations while our work defines behavioral semantics basing on proved B modeling operations (like the constructor of Fig. 5) which guarantees that our behavioral semantics always preserve the invariant properties of the meta-model.

4 Modeling

There exist several tools dedicated to the instantiation of meta-models. Among those based on EMF one can use: Sirius,[7] EuGENia,[8] GMF,[9] etc. These tools associate graphical representations to concepts issued from a meta-model and

[7] https://www.obeo.fr/fr/produits/Eclipse-sirius.

[8] https://www.Eclipse.org/epsilon/doc/eugenia/.

[9] http://www.Eclipse.org/modeling/gmp/.

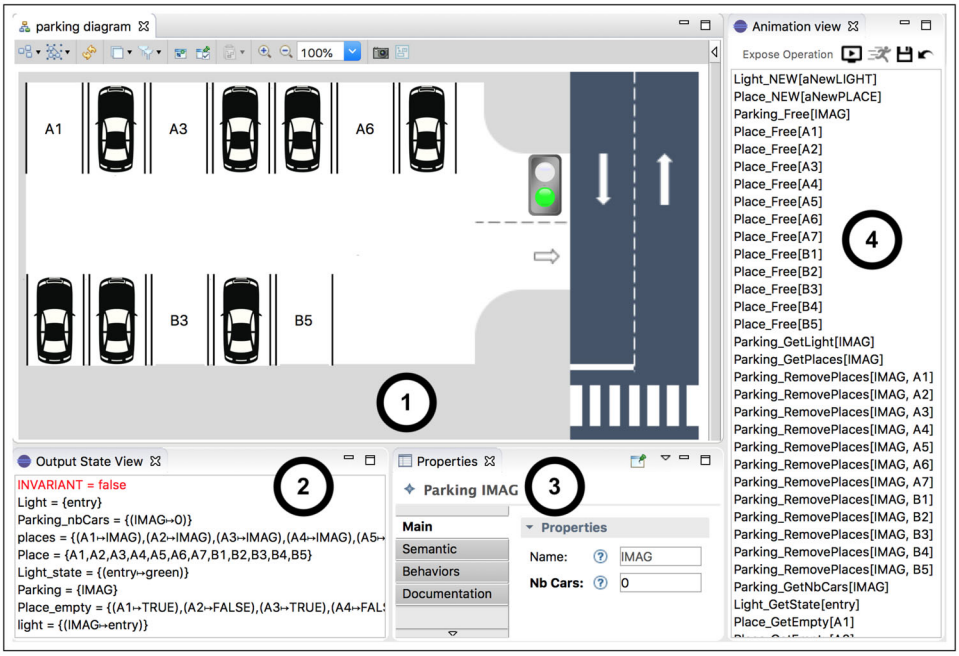


Fig. 8 A screenshot of the parking modeling view

then they allow to draw graphical models which are readable and intuitive from the end-user point of view. Figure 7 gives the palette of representations used in our DSML to represent lights and places depending on their states.

4.1 Modeling environment

The modeling environment including the Meeduse tool is presented in Fig. 8. This environment is structured into four Eclipse views: graphical view (1), output state view (2), properties edition view (3) and animation view (4). The first two views represent the model itself in two different syntaxes: a graphical syntax (view 1) and the B syntax with enumerated sets (view 2). The two other views allow model edition in two different ways: using the EMF editor plugin (view 3) and using the B operations (view 4). Views 2 and 4 are then the additional views offered by Meeduse to a classical EMF-based modeling.

The modeling view allows the designer to draw a parking model in a domain specific syntax. For this example the graphical concrete syntax (used in Fig. 8) for the parking DSML was defined using the Sirius tool. However, any other modeling tool can be used together with Meeduse since it instantiates an EMF resource representing the model. The main advantage of Sirius in comparison with other EMF-based modeling tools is its facility to define conditional styles with an OCL-like syntax. For example, the representation of a light would depend on the evaluation of the following constraints which are specified in context of class Light: "`self.state = Color::green`" and "`self.state = Color::Red`". The model of Fig. 8

$$
\begin{aligned}
&Parking := \{IMAG\}\\
&Light := \{entry\}\\
&Place := \{A1, A2, \ldots\}\\
&light := \{(IMAG \mapsto entry)\}\\
&places := \{(A1 \mapsto IMAG), (A2 \mapsto IMAG), \ldots\}\\
&Light\_state := \{(entry \mapsto green)\}\\
&Place\_empty := \{(A1 \mapsto TRUE), (A2 \mapsto FALSE), \ldots\}\\
&\ldots
\end{aligned}
$$

Fig. 9 B initialization issued from a given EMF model

shows an example of a parking with twelve places, where five places are empty, and a light set to green.

In the following, we discuss the integration of Meeduse into an EMF-based modeling environment, by showing how the illustrative model of Fig. 8 is translated into B valuations and vice-versa.

4.2 From an EMF model to B valuations

The output state[10] view in Fig. 8 shows the B machine state by means of B variables valuations which are issued from a given EMF model. Meeduse injects the various model elements into the B specification produced from the meta-model using the initialization clause. For example, in this model the value of attribute $state$ associated to the light named "entry" in parking "IMAG" is equal to $green$, giving rise to the initialization of variables $Parking$, $Light$, $light$ and $Light\_state$ as presented in Fig. 9.

[10] The notion of state, in the B method, corresponds to valuations of B variables.

After generating the valuations from an EMF model, Meeduse calls the model-checker ProB [26] in order to animate the corresponding initialization and also in order to verify whether the invariant is violated or not by the initial state of the B machine. Then, Meeduse visualizes both this state and the information about invariant preservation in the output state view. Figure 8 shows an invariant violation due to the value of attribute $nbCars$ which can be set by EMF operations without any verification. Indeed, we showed that the use of EMF-based modeling tools (like Sirius for example) is error prone for model construction because of the permissiveness of the Java code generated from the meta-model. When the user creates a parking via an EMF-based modeling tool, then the tool implicitly calls the Java constructor discussed in Fig. 3. The inconsistency between the initial value of attribute nbCars and the OCL constraint was not detected by EMF during the meta-modeling step and hence the modeling view creates a wrong EMF model were this attribute is initialized to zero and can also be set to any integer value using the properties view of EMF.

Hopefully, EMF addresses this limitation by offering a verification function which discovers errors of the model with respect to OCL constraints and the structural properties of the meta-model. However, this big-bang verification has several drawbacks for more complex and realistic models: (i) it can be time consuming because it does not give enough information about what goes wrong during the modeling activity, and (ii) it can slow down productivity because it may require improvements of the meta-model with its constraints and hence may have impact on the generated Eclipse plugins like model editors.

4.3 From B valuations to an EMF model

In the previous sub-section, we presented a model designed by applying the default EMF editor, and we showed how this model can be translated as valuations of the B variables in order to check its validity with respect to the meta-model invariants. In order to avoid an a-posteriori verification based on a specific given EMF model, our proved B operations can be used. Indeed, these operations guarantee the construction of a correct model all along its design. For example, starting from an empty model were all B variables are equal to the empty set, the following sequence of basic modeling operation calls constructs a parking (MyParking) with one empty place (P1), one occupied place (P2) and a light (myLight) whose state is set to $green$ by default:

```
1. Light_NEW(myLight);
2. Place_NEW(P1);
3. Place_NEW(P2);
4. Parking_NEW(MyParking, myLight, {P1, P2});
5. Place_SetEmpty(P2, FALSE);
```

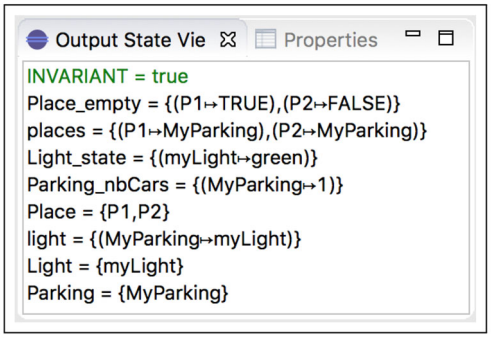


Fig. 10 Reached B state

Figure 10 gives the reached B machine state after the animation of the above sequence of B operations. obviously, the invariant is kept true during this scenario leading then to a model which is correct by construction. Manipulation of models using the B formal model issued from the meta-modeling activity can be done step by step thanks to the animation view. This view lists all B operations which can be enabled from the current state of the B machine. The enabledness of these operations is computed by the ProB model-checker after evaluating the preconditions of every operation. For example, from the empty state, the only possible operations that can be enabled are Light_NEW and Place_NEW. Operation Parking_NEW can be enabled only when the model contains a light and at least one place according to the precondition provided in Fig. 5. Also, operation Place_SetEmpty(P2, FALSE) is enabled only from a state where place P2 is linked to a Parking, which is done in step 4.

The usage of the proved modeling basic operations allows to keep the model correct all along its creation while a classical EMF-based approach requires an additional validation after the model creation in order to check its correctness. In addition to this step-by-step animation for model construction, it is also possible to give a pre-established sequence of operations and ask Meeduse to animate it and then get the final model. In both cases, after every animated operation, the tool gets from ProB the new state of the B machine and translates it back to the EMF model. The non-feasibility of a sequence of operations means that it includes a call to an operation in a state which is not compatible with its precondition.

4.4 Foundations of the mapping from B to EMF

A state in the B method represents a particular valuation of B variables. Since Meeduse produces these variables from the meta-model, then a B machine state should be equivalent to a given instance of the meta-model. In order to manage this equivalence between the EMF model and the B machine

state a constraint solving technique is proposed where two main objectives are covered: creation/destruction of objects, and setting/unsetting of attributes and relationships between these objects. Taking into account states before and after the animation of a B operation, the proposed technique identifies the right sequence of EMF operations called by Meeduse in order to modify the EMF model according to the difference between these states. Giving a state variable var in the B machine, we call:

- $\mathcal{M}(var)$: the meta-modeling element from which variable var was mapped during the transformation from the Ecore meta-model into B. For example, $\mathcal{M}(Light\_state)$ means attribute state and $\mathcal{M}(Light)$ means class Light.
- $EType(var)$: the Ecore type of $\mathcal{M}(var)$, such that

$$EType(var) \in \{\text{EClass, EAttribute, EReference}\}.$$

  For example, $EType(Light\_state) =$ EAttribute.
- $Nature(var)$: the arity of an attribute or a reference defined by:

$$Nature(var) \in \{\text{Object, List}\}, \quad \text{if } EType(var) \neq \text{EClass}$$

  For example, $Nature(Light\_state) =$ Object, since attribute state is mono-valuated and $Nature(places) =$ List because reference places is multi-valuated.

Considering $S(var)$ and $S'(var)$, the sets of valuations of variable var, respectively, before and after the animation of a B operation (or a sequence of B operations); and val a given value of var such that $val \in S(var) \cup S'(var)$, then we define:

- $Obj(val)$: a function giving the object which corresponds to value val. For example, $Obj(P1)$ gives from the EMF model, object P1 instance of class Place; and $Obj((myLight \mapsto green))$ gives object myLight instance of class Light.
- $Elem(val)$: the mapped element if val is a pair. For example, $Elem((myLight \mapsto green))$ gives the literal green.

4.4.1 State before/after difference

Having these definitions, our approach basically computes two formulas:

$$F_1(var) \mathrel{\hat{=}} S_{var} - S'_{var} \text{ and}$$
$$F_2(var) \mathrel{\hat{=}} S'_{var} - S_{var}.$$

Formula F_1 computes the set of values that have been removed from one state to another and F_2 computes the added values. For example, considering the following valuations which are reached before and after the animation of operation Place_SetEmpty(P2, FALSE):

$$S_{\text{Place\_empty}} = \{(P1 \mapsto TRUE), (P2 \mapsto TRUE)\}$$
$$S'_{\text{Place\_empty}} = \{(P1 \mapsto TRUE), (P2 \mapsto FALSE)\}$$

then, F_1(Place_empty) = {(P2 $\mapsto$ TRUE)} and F_2(Place_empty) = {(P2 $\mapsto$ FALSE)} which means that value $(P2 \mapsto TRUE)$ is removed and value $(P2 \mapsto FALSE)$ is added.

4.4.2 Mapping rules

Depending on functions $EType$ and $Nature$ we apply the following rules to extract EMF operations for every value val:

Case 1: $EType(var) =$ EClass

1. rule 1.1 ($val \in F_1(var)$):

```
EcoreUtil.delete(Obj(val));
```

2. rule 1.2 ($val \in F_2(var)$):

```
M(var) val = factory.createM(var)();
val.setName("val");
eResource().getContents().add(val);
```

When variable var is produced from a class in the meta-model, then rule 1.1 deletes the selected instances of this class from the EMF model using the Java EcoreUtil API, while rule 1.2 creates the objects. This second rule first creates an object using the constructor generated by EMF (such as operation createParking of Fig. 3), then associates a name to this instance assuming that the meta-model deals with named objects and finally, it adds the object to the EMF resource.

Case 2: $EType(var) \in$ {EReference, EAttribute} and $Nature(var) =$ Object

1. rule 2.1 $val \in F_1(var)$ and $\forall val' \cdot (val' \in F_2(var) \Rightarrow Obj(val) \neq Obj(val'))$:

```
Obj(val).setM(var)(null);
```

2. rule 2.2 ($val \in F_2(var)$):

```
Obj(val).setM(var)(Elem(var));
```

State before

$Parking = \emptyset$
$Light = \{myLight\}$
$Place = \{P1, P2\}$
$light = \emptyset$
$places = \emptyset$
$Parking\_nbCars = \emptyset$
$Light\_state = \{(myLight \mapsto green)\}$
$Place\_empty = \{(P1 \mapsto \mathbf{TRUE}), (P2 \mapsto \mathbf{TRUE})\}$

State after

$Parking = \{MyParking\}$
$Light = \{myLight\}$
$Place = \{P1, P2\}$
$light = \{(MyParking \mapsto myLight)\}$
$places = \{(P1 \mapsto MyParking), (P2 \mapsto MyParking)\}$
$Parking\_nbCars = \{(MyParking \mapsto 0)\}$
$Light\_state = \{(myLight \mapsto green)\}$
$Place\_empty = \{(P1 \mapsto \mathbf{TRUE}), (P2 \mapsto \mathbf{TRUE})\}$

Fig. 11 Before and after states

Rules 2.1 and 2.2 address single-valuated structural features. It assigns null to the feature when an existing valuation is removed, or updates it to the mapped value $Elem(var)$. For example, from the previous valuations $S(Place\_empty)$ and $S'(Place\_empty)$ rule 2.2 is triggered because P2 is in both F_1(Place_empty) and F_2(Place_empty), meaning that the value of feature empty is overwritten by a new value. Then, the following call to the basic setter of attribute empty in meta-class Place, on object P2 is called by Meeduse: `P2.setEmpty(FALSE);`

Case 3: $EType(var) \in$ {EReference, EAttribute} and $Nature(var) =$ List

1. rule 3.1 ($val \in F_1(var)$)

$$Obj(val).\texttt{get}\mathcal{M}(var).\texttt{remove}(Elem(var));$$

2. rule 3.2 ($val \in F_2(var)$)

$$Obj(val).\texttt{get}\mathcal{M}(var).\texttt{add}(Elem(var));$$

Rules 3.1 and 3.2 apply a similar technique as the previous rules, but on multi-valuated structural features. They result in the use of operations remove and add on Java lists.

Discussion. In order to illustrate these rules, we consider in Fig. 11 states before and after the animation of operation Parking_NEW(MyParking, myLight, {P1, P2}).

Having these valuations, Meeduse computes automatically the following sequence of EMF operations. Despite the fact that the intermediary steps of this sequence may lead to erroneous models, the model obtained at the end is obviously conformant to the invariant. This conformance is guaranteed by the B machine state and its mapping into EMF as presented in this section.

```
1. Parking MyParking = factory.createParking();
2. MyParking.setName("MyParking") ;
3. eResource.getContents.add(MyParking);
4. MyParking.setLight(myLight);
5. MyParking.getPlaces.add(P1);
6. MyParking.getPlaces.add(P2);
7. MyParking.setNbCars(0);
```

5 Model execution

The intention of our approach with its tool support is to put into practice an alliance between a formal method and MDE when designing a domain-specific modeling language like that of the parking. Sections 3 and 4 presented the modeling aspects of this alliance. We showed that the formal B specification allows automated reasoning about the correctness of the construction operations issued from a meta-model, and provides an efficient way to keep a correct model during the design activities. This section addresses the executional aspects of a model by showing how the B specification produced from the previous steps can be enhanced in order to define how a given model should behave. In our approach, these operational semantics are structured into several concerns: abstract behavior, refined behavior and concrete behavior. For the parking lot example, the concrete behavior would be that of a targeted embedded controller which acts on the traffic light in order to allow or deny the access depending on place availability.

5.1 Abstract behavior

The abstract behavior defines the useful operations for the targeted system together with their sequencing. The target system of our parking lot is an embedded controller that applies four operations: enter and leave which would be called by the environment (in this case we refer to the physical entrance/leaving sensor), and control_entry and control_exit which are internal operations triggered by the controller after each entry and exit. Figure 12 shows a state/transition diagram that represents the intended sequencing of these operations. The corresponding B machine, called abstract_behavior, defines a variable named $state$ in order to represent the different states of this diagram. This variable is initialized to $waiting$ and gets values $entering$ or $leaving$, when the various transitions are triggered:

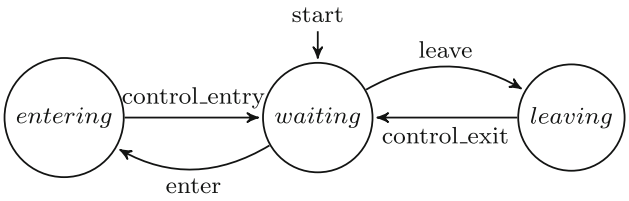


Fig. 12 Abstract behavior

enter = SELECT state = waiting THEN state := entering END

leave = SELECT state = waiting THEN state := leaving END

control_entry = SELECT state = entering THEN state := waiting END

control_exit = SELECT state = leaving THEN state := waiting END

Note that our parking lot controller is completely synchronous and depends on when it receives messages from the physical entrance/leaving sensor. In fact, if the sensor captures an entrance and a leaving at the same time, it may simply forget the signal which remains a safe behavior because one entry and one leaving at the same time mean for the controller that the number of cars is not changing. Another possibility is to introduce an additional operation without any effect, like for example: enter_leave = BEGIN skip END. This operation can be illustrated on Fig. 12 by a reflexive transition on state waiting. If the sensor gets an entrance and a leaving at the same time, it has to call enter_leave, and in all cases this would be without any effect on the number of cars.

This behavior is called abstract because it does not deal with modeling data structures like the availability of places and the permission to enter or leave the parking lot. This information will be introduced incrementally thanks to the refinement mechanism in B. At this stage, the abstract behavior means that operations enter and leave must be controlled before any other entry or exit.

5.2 Refined behavior

As presented in Fig. 1, the operational semantics applies two mechanisms: refinement and inclusion. It refines the abstract behavior and includes the B machine generated from the meta-model. Refinements in the B method have three main objectives : (i) gradually adding requirements, (ii) going from abstract notions to more concrete ones until getting at the implementable code level, and (iii) ensuring development correctness since each refinement has to preserve the properties of the component it refines. The composition mechanism, such as inclusion, allows to break down a system into smaller parts where the separation of concerns principle is applied. Our intention is then to preserve the sequencing of Fig. 12 by refinement, and to use data and operations describing the model by inclusion. We introduce in the corresponding B machine, a variable called $thePark\mathit{ing}$ defined as $thePark\mathit{ing} \in Park\mathit{ing}$ and initialized by $thePark\mathit{ing} :\in Park\mathit{ing}$ (*i.e.,* theParking takes some value from set Parking). The operational semantics is then applicable to any parking object provided the five following safety properties.

In the waiting state, the number of cars inside the lot must be less than the total number of places when the light is green, otherwise (the light is red) the number of cars must be equal to the number of places. These properties guarantee that entry to the parking lot is allowed (green light) only if there exists at least one empty place, and it is forbidden (red light) when the maximum capacity is reached.

Properties 1 and 2

$$\begin{array}{l}((state = \text{waiting} \wedge Light\_state(light(theParking)) = green) \\ \quad \Rightarrow Parking\_nbCars(theParking) < \mathbf{card}(places^{-1}[\{theParking\}])) \\ \wedge\ ((state = \text{waiting} \wedge Light\_state(light(theParking)) = red) \\ \quad \Rightarrow Parking\_nbCars(theParking) = \mathbf{card}(places^{-1}[\{theParking\}]))\end{array}$$

When variable state, which represents states of our state/transition diagram, is equal to entering, the light must be green (*i.e.,* value of attribute state in class Light is equal to green). Indeed state waiting in the state/transition diagram is reached after one entry and it is operation control_entry that switches the light from green to red.

Property 3

$$(state = \text{entering} \Rightarrow Light\_state(light(theParking)) = green)$$

Finally, in state leaving, when the light is red, then only one place was freed due to the leave operation which happens when the parking lot is full. And, if the light is green then there is more than one empty place.

Properties 4 and 5

$$\begin{array}{l}((state = \text{leaving} \wedge Light\_state(light(theParking)) = green) \\ \quad \Rightarrow Parking\_nbCars(theParking) < \mathbf{card}(places^{-1}[\{theParking\}]) - 1) \\ \wedge\ ((state = \text{leaving} \wedge Light\_state(light(theParking)) = red) \\ \quad \Rightarrow Parking\_nbCars(theParking) = \mathbf{card}(places^{-1}[\{theParking\}]) - 1)\end{array}$$

The behavioral part of our operational semantics is specified by refining operations of Fig. 12 and by proving the safety properties. The sequencing defined by the abstract behavior is also preserved by proofs. For example, the following refinement of operation enter takes into account the green light imposed by the third property and applies operation Place_SetEmpty (presented in Fig. 6) on a place chosen non-deterministically from the set of empty places. The transition from state $waiting$ to state $entering$ is preserved.

```
enter =
  SELECT
    state = waiting ∧ Light_state[light[{theParking}]] = {green}
  THEN
    ANY place WHERE
      place ∈ Place ∧ place ∈ places ⁻¹ [{theParking}]
      ∧ Place_empty[{place}] = {TRUE}
    THEN
      Place_SetEmpty(place, FALSE)
    END ;
    state := entering
  END;
```

The non-deterministic substitution of operation enter (clause ANY *place* WHERE…) means that when a car enters, a place is immediately assigned to it no matter which one. At a high level this is sufficient to guarantee the safety invariants of the parking lot. The duration between the car entering and the place assignment can be added by refinements, and hence it would be an additional specification. Nonetheless, for this running example, we choose to keep the specification as simple as possible.

Operation enter is triggered only if the light is green, the system controller is in state *waiting* and there exists at least one empty place. Operation control_entry can then be done by the controller in order to switch the light to red when the parking lot is full. Conforming to the abstract behavior control_entry returns the system to state *waiting*.

```
control_entry =
SELECT state = entering THEN
  IF
    Parking_nbCars(theParking)
      = card(places ⁻¹ [{theParking}])
  THEN
    Light_SetState(light(theParking), red)
  END;
  state := waiting
END;
```

Operations leave and control_exit are similar to enter and control_enter. For place reasons, they are not presented. The guard of operation leave is $state = waiting \wedge Parking\_nbCars(theParking) > 0$ which means that the operation is triggered if there is at least one car in the parking lot. Then, its actions free a place non-deterministically and go to state *leaving*. control_exit enabled from state *leaving*, switches the light from red to green, if the light is red. Indeed, when a car leaves the parking lot then a place has been freed as required by properties 4 and 5.

Discussion. This sub-section presented an example of a possible refined behavior where no parallel assignment of places is taken into account. Indeed, other possible behaviors can be applied to a set of places (not only one as proposed by operation entry) in order to change their status simultaneously. This would be an other B specification that must also be proved correct regarding the parking lot safety invariants. The main question is then: how to validate which behavior is the most suitable from the expert point of view? In fact, the high level behavior, even if it is safe (by proofs), maybe not accepted by domain experts. The interest of Meeduse is that it makes these behaviors observable over the model itself by applying animation. For the refined behavior specification, the AtelierB prover generated 82 POs that it was able to prove automatically. These POs focus only on the safety properties since associated operations call basic operations such as Light_SetEmpty which are already proved correct with respect to the meta-model invariant. Like in Fig. 8, Meeduse allows to load this B specification in order to animate its operations leading to automatic modifications of the EMF model. The animation of the operational semantics in Meeduse allows the end user to play and then validate the corresponding behavior by visualizing its effects on the model.

5.3 Concrete behavior

The transition from a given model done using a DS(M)L into a target platform remains an open research direction. In MDE, this may be done by applying transformation rules which are defined over meta-models and executed on the model. However, the challenging requirement when applying transformations is to guarantee that the resulting platform behaves like expected by the model. Nonetheless, often the targeted system uses concepts which are far from those used in the modeling language. For example, the physical parking controller device does not manage instances of classes with associations, and does not have any knowledge about the places. It needs to know the parking lot capacity (called *cap*), and it manages the authorization (called *auth*) and an internal counter (called *cars*) that it increments/decrements according to the entrances and exits.

In our approach we take benefit of proved B refinements in order to guarantee that the embedded parking control system preserves the model behavior defined by the operational semantics and validated by the end-user. We introduce a linkage invariant which defines concrete variables in relation with those issued from the meta-model:

```
auth ∈ 0 . . 1 ∧ cap ∈ NAT ∧ cars ∈ NAT
∧ (auth = 0 ⇒ Light_state[light[{theParking}]] = {green})
∧ (auth = 1 ⇒ Light_state[light[{theParking}]] = {red})
∧ cap = card(places ⁻¹ [{theParking}])
∧ cars = Parking_nbCars(theParking)
```

Based on this linkage invariant, operations issued from the parking lot operational semantics are redefined by means of concrete variables. The concrete behavior specification is presented in Fig. 13.

```
enter =
  SELECT
    state = waiting ∧ auth = 0
  THEN
    cars := cars + 1 ;
    state := entering
  END;
```

```
control_entry =
  SELECT state = entering THEN
    IF cars = cap THEN
      auth := 1
    END ;
    state := waiting
  END;
```

```
leave =
  SELECT
    state = waiting ∧ cars > 0
  THEN
    cars := cars - 1 ;
    state := leaving
  END;
```

```
control_exit =
  SELECT state = leaving THEN
    IF auth = 1 THEN
      auth := 0
    END ;
    state := waiting
  END
```

Fig. 13 Specification of the concrete behavior

Discussion. The concrete behavior does not impact the model, because it represents the final embedded system. For example, operation enter in the concrete behavior (Fig. 13) does not refer to any meta-modeling element (Place, Light, Parking, etc) contrary to its specification in the abstract and refined behaviors. The preservation of the structural invariant properties defined at the meta-modeling level (Sect. 3) and also the invariants that guarantee the safety of the behavioral semantics is ensured with respect to the linkage invariants. In fact, linkage invariants in B allow to formally specify the relationship between the various refinement levels and hence the associated proof of correctness mean that concrete operations preserve the invariant properties of the abstract level. As a consequence, we have the guarantee that the concrete behavior, even if it does not impact the model, satisfies the properties of the model as well as those of the operational semantics. For this last step, the AtelierB prover produced 58 OPs allowing to check the conformance of this concrete behavior with respect to the operational semantics. The above specification is further refined into an implementation from which the AtelierB produced automatically a C program for our parking lot controller. Thanks to refinements, proofs and automatic code generation allowed by AtelierB this program behaves as described in Fig. 12. Then for every execution step, the C program can generate a trace which refers to the executed function and then this trace can be replayed by Meeduse on the EMF model (by component trace runner). This technique animates the model during program execution which is useful for conformance validation or runtime verification.

6 State of the art

6.1 UML-based techniques

Several works [15,22,24,25,32–34] have been devoted in order to establish the relationship between a formal approach and model driven engineering. Most of these works deal with automatic transformation of UML diagrams (mainly class diagrams and state/transition diagrams) into a formal language in order to use proof and model-checking tools. Other works tried to go a step further towards model implementation in targeted technologies. In [6] the authors used the Z notation to show how abstract class models of an information system can be translated into more concrete designs, and then compiled into a working implementation. A similar idea was proposed in [27] were the B language was investigated for data-base generation from UML diagrams. These techniques cover the whole development process since they start from models and lead to implementations by successful proved refinements. Nevertheless, the reverse link (from concrete design to abstract models) is missed because they do not take into account model execution at early design stages. In our approach, we introduce animation and trace running during the development process in order to keep the behavior conformance from modeling to implementation.

Furthermore, UML-based techniques do not address domain-specific modeling languages (DSML) because they are limited by the UML notations and semantics. Indeed, UML requires additional semantics to be able to describe in a precise way all the conceptual constructs related to a domain-specific model-driven development. For example, if Fig. 2 is assumed to be a UML class diagram by these works, then its instances should be represented by UML object diagrams which are, on the one hand, less intuitive from the end-user point of view than the screenshot of Fig. 8, and on the other hand, semantically far from the domain knowledge. Our proposal circumvents this shortcoming thanks to the use of DSMLs rather than UML extensions.

As far as we know, only two publicly available tools for the linkage between UML and the B method exist: iUML-B [8,35], and B4MSecure [15] that we developed in our previous works. In Meeduse, the extraction of B specifications from a meta-model applies a first transformation from ECore to UML, and then a second transformation from UML to B. This second transformation is ensured thanks to B4MSecure whose advantage in comparison with other approaches is that it offers an extensibility facility allowing to easily add new UML-to-B rules or to modify existing rules depending on the application context. In this paper, the application context is that of meta-models and DS(M)Ls. This mechanism makes the extraction of the formal specifications more flexible than a one to one transformation like proposed in the existing works. For example, the current B specification of the constructor of class parking (Fig. 5) requires to have already created its light and its places, and only once they exist, one can create also a parking. Nonetheless other transformations may be interesting like for example, a constructor that creates the parking together with its light and its places. In this case, the extension mechanism of B4MSecure is interesting because it allows to easily integrate this new transformation

within the currently encoded catalog of transformations. The main intention is to provide to the designer an adaptable and flexible transformation that produces a B specification from which one can prove properties of the data structures independently of functionalities to be implemented contrary to existing works which often mix both concerns.

6.2 Domain-specific techniques

The use of a formal specification provides several benefits, in comparison with EMF-based techniques that deal with operational semantics [3,19,36]. The major one is that V&V activities can be dissociated in several concerns: properties of the meta-model, those of the behavior and finally those of the target platform. The EMF-based approaches like Kermeta [19], XCore [37], etc, suffer from limitations of EMF which are discussed in Sects. 3 and 4. Then, the major V&V activities proposed by these approaches are done by simulations and testing. Besides the use of automatic reasoning tools (AtelierB and ProB), this work investigated the use of B refinements in order to gradually translate an expected behavior into a concrete one which can then be automatically compiled into a programming language without missing the relationship with the initial EMF model.

Few proposals [11,29] exist for bridging the gap between DSMLs with their supporting tools and a proof-based formal approach. Some formalisms are used like LOTOS, MARTE or abstract state machines (ASM), but their main shortcoming is that once the translation is done, they do not offer any way to animate jointly the formal model and the EMF model. The proposed development often starts from a meta-model and then gets lost in the formal process. This paper addressed this point in Sect. 4.4 showing technically how to go back from a formal state to an EMF model with its concrete syntax all along the formal process. In [45], the authors propose to provide the system designer with a user-friendly language (FIACRE) to formalize system requirements. However, the associated formal verification is done through model-checking which suffers from the well-known limitations: bounded state space and state explosion problems. Our approach contributes to these works using automated theorem proving from the meta-modeling activity to the final system generation.

More recently, in [39–41] the authors applied Event-B in order to establish a predefined library of proved specification templates that can be composed by the DSL designer at a high level. The advantage of these semantical templates is that they may facilitate the tasks of the language designer when modeling the semantics of a DSL, but they also suffer several restrictions compared to what can be done using a formal language like B. In this paper, we keep some freedom when using the B language constructs for the formal DSL definition. We believe that when a formal method is applied, skills in both formal methods and model-driven engineering are required, even with a pre-established catalog of semantical templates. Indeed, in order to cover a wide range of DSLs, new templates may be required, or some existing templates must be adapted and proved, etc. In our proposal, we advocate for collaborations between both communities (FM and MDE) in order to take benefits of their complementarities. The Meeduse tool described in this paper favors this collaboration since it makes possible the use of MDE and formal methods tools together in one unified framework. The Meeduse tool manages the synchronization and the traceability between the EMF artifacts and the formal specifications. The use of ProB to animate the formal operational semantics results in an automatic visual animation of the domain model directly in the DSL tool. Contrary to our approach, [41] applies classical visual animation by using a visualization tool (BMotion Studio) to the formal specifications. Unfortunately, this is time consuming because the graphical representations must be redefined in BMotion Studio. Moreover, it also requires some additional verifications in order to address the compatibility between the initial DSL syntax and the graphical visualization done by BMotion Studio.

7 Application

In this work, the DSL tool is developed in a well-known model-driven engineering paradigm (MDE) based on EMF and it is intended to be used by the domain expert. The formal part of our approach is assisted by Meeduse which automatically translates the DSL meta-model into an equivalent B specification and then the operational semantics of the DSL are defined using proved B operations that guarantee the correctness of the model's behavior with respect to its invariant properties. This approach was successfully applied on realistic case studies and showed by practice several benefits of this alliance between MDE and a proof-based formal approach such as the B method. In the following, we describe two real-life applications of Meeduse and we discuss their results.

7.1 A formal railway DSML

In [16,17], we present the application of our approach to a realistic industrial case study issued by the railway domain. Figure 14 is a screenshot of the resulting DSL tool which allows the domain expert to define railway topologies (bottom side of the figure) with fine grained views (middle side and top side of the figure). Like for the parking lot example, the graphical modeling elements are designed in Sirius. Regarding the operational semantics of this DSL, they were defined using B operations that allow train movements, routes assignments to trains, modifications of switches positions,

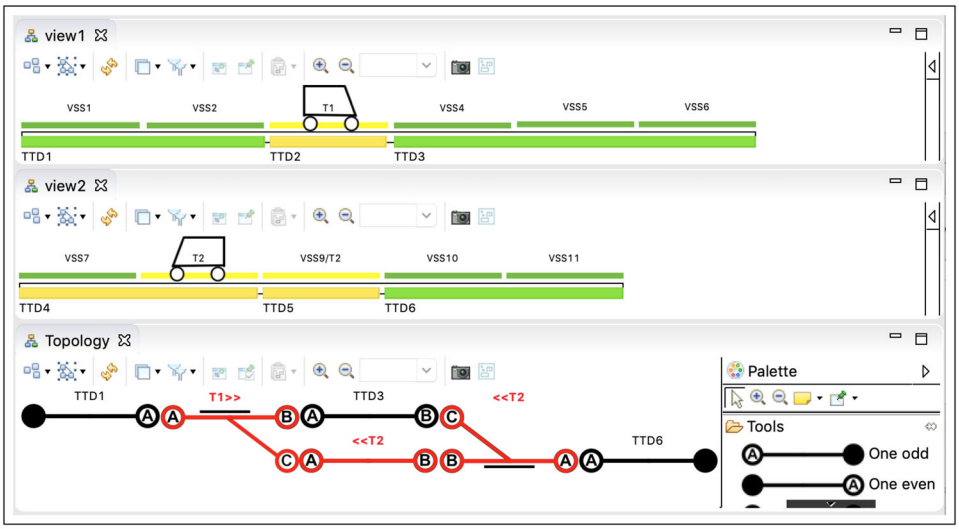


Fig. 14 Application of Meeduse to a railway case study

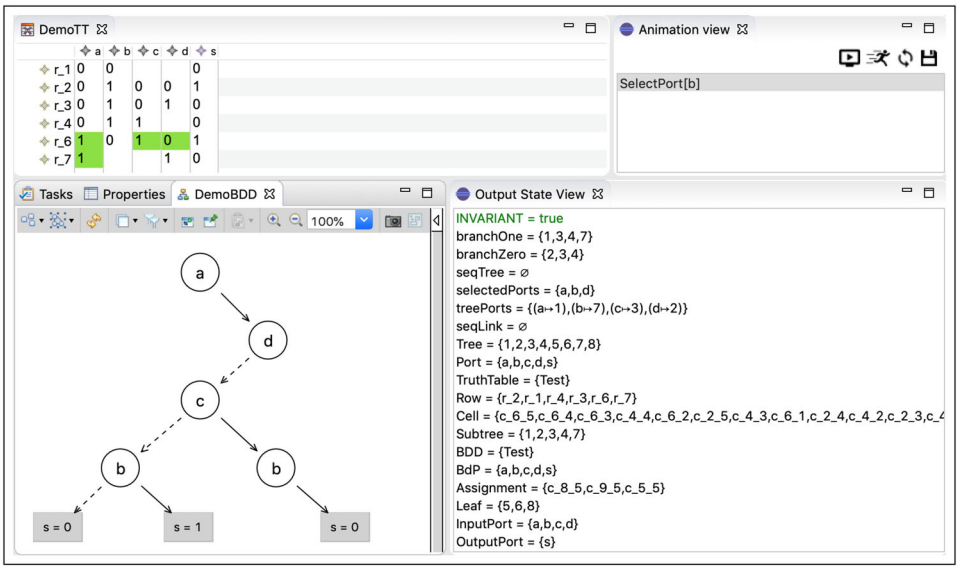


Fig. 15 Application of Meeduse to DSL transformation

etc. The proof of correctness of these operations guarantees safety behaviors where no accidents may happen.

This application of Meeduse to a railway case study is interesting because of two main observations: first, in railway control and safety systems the application of formal methods is a strong requirement; and second, graphical representations of domain concepts are omnipresent in railway documents thanks to their ability to share standardized information with common knowledge about several mechanisms: track circuits, signalling rules, etc. Meeduse showed its strength to mix both aspects in a tool that can be used by railway experts who are often not trained neither in MDE nor in FM techniques.

7.2 A formal DSL transformation

Meeduse was also applied to formally define DSL transformations based on a real-life case study which is that of translating truth tables into binary decision diagrams. This application [18] carried out during the 12th edition of the transformation tool contest[11] (TTC'19) won the award of best verification and the third audience award. Figure 15 shows the various views of Meeduse where the Sirius modeling deals with the representation of the input DSL. In this application, formal operational semantics, written in B, consume truth table elements and progressively produce a binary decision diagram. This application allowed us, on the one hand, to figure out how far we can push the abilities of a formal method to be integrated within model-driven engineering, and on the

[11] https://www.transformation-tool-contest.eu.

Table 1 Proof obligations generated from the railway DSML

	Proof obligations	Automatic	Manual
Level 1	17	11	6
Level 2	32	25	7
Level 3	62	41	21

other hand, to present an original approach to think, reason and execute DSL transformations using a formal method.

7.3 Results

The interest of the railway DSML is that it deals with several safety-critical invariants for which theorem proving are required in order to guarantee a zero-fault behavior. The length of the B machine generated from the meta-model is about 500 lines with 38 primitive operations from which the AtelierB produced 80 proof obligations that were proved automatically. Regarding the operational semantics of this DSML they were introduced incrementally using three proved refinement levels. The numbers of proofs generated from these refinements are presented in Table 1. In general, proving a specification is a difficult and time-consuming task which is the reverse side of formal methods; but we think that for this case study these proofs remain manageable and of reasonable complexity, especially for B method experts.

In the DSL transformation case study, the B machine extracted from the meta-model is about 1162 lines of code from which the AtelierB generated, and automatically proved 260 proof obligations. This proof gives the guarantee that the model's properties are preserved during the DSL transformation. The operational semantics were introduced using an additional B machine whose length is about 150 lines of code. This is more concise (but may be more difficult to understand) than a reference ATL solution which is about 340 lines of ATL code. The proposed operational semantics of this case study were defined through five B operations and several additional invariants. For this case study the correctness of the operational semantics was ensured by model-checking, rather than by theorem proving because on the one hand it is less time consuming, and on the other hand, it deals with bounded state spaces that can be exhaustively checked by the ProB model-checker.

8 Conclusion

This paper has shown, based on a simple example, practical aspects of the integration between a proof-based formal approach and MDE. The objective is to apply proofs and refinements during the model-driven development process. Our approach is supported by a tool-set called Meeduse that allows: (i) translation of an Ecore meta-model into a B machine in order to prove its correctness, (ii) injection of a model into this B Machine in order to check its consistency with respect to the meta-model properties, (iii) animation of the B specification with associated operational semantics in order to validate its behavior from the end-user point of view, and (iv) running a given execution trace for conformance and runtime verification.

Our model-driven software development process combines EMF-based metamodeling and instance animation with formal verification and stepwise refinement in B (with the help of the AtelierB and ProB tools). The overall approach follows four steps:

1. A meta-model (close to a UML class diagram) created using EMF/Ecore is translated automatically into corresponding declarations and type invariants in B where the type correctness can be proved formally using theorem proving techniques.
2. The user then adds B specifications for operations (i.e., pre/post-conditions) and additional constraints (such as 'class invariants') to which the B tools are applied to establish correctness.
3. B refinement techniques are used to equip the model with provably correct execution semantics.
4. The C code generator in the AtelierB tool is used to generate code and collected traces are replayed on the EMF model.

Connecting B and MDE is mutually beneficial because the B method is a mature methodology with strong foundations, many extensions, good tool support, and a significant user community. In this paper, we showed the interest of a proof-based meta-modeling in comparison with a classical technique where a Java source code is produced from a meta-model and we provided a step-by-step construction of a DSL's dynamic semantics starting from the most abstract behavior to the most concrete one. We applied our approach to two real-life applications and we discussed their results in this paper. We have also experimented Meeduse on several case studies: Petri-nets DSL, automatic car light regulator, process scheduler, tic-tac-toe, puzzle game, lift, family model, etc. Demonstration videos of our tool and its application to graphical and textual DSL animation can be found at: http://vasco.imag.fr/tools/meeduse/.

The major limitation of this alliance of MDE with a formal method is that it requires a high background in both MDE and formal methods in order to be applied. However, in our methodology we do not advocate that an MDE expert must become expert in the B method in order to apply Meeduse, but we advocate for a collaboration between MDE experts

and formal methods experts. We believe that DS(M)Ls interest the safety-critical community whose main intention is to develop systems which are bug-free because their failures would lead to human loss. Meeduse gives solutions in this direction: the DSL tool development is the task of MDE experts who have the ability to define meta-models with associated static constraints in EMF, and the formal semantics specification is the task of the formal methods experts who are able to apply provers and model checkers.

This work has several perspectives mainly about how to use the potential of formal and MDE tools when both paradigms are mixed together:

- Formal tools: In this paper, we limited our study to refinements and proofs; however, several other formal techniques can be of a great interest for MDE, like constraint solving or model-checking. Once the formal specifications are written and proved, these techniques can be applied in order to provide a way for reachability analysis. Indeed, when defining operational semantics the designer often needs to quickly get a model satisfying some properties or to get a sequence of operations leading to such a model. For example, one interesting state is that where the parking lot is full. Instead of filling it manually by animation or by setting attribute empty to false for every place, a constraint solver could produce a solution satisfying the required property, and a model-checker could produce the sequence that reaches this state.
- MDE tools: The study presented in this paper addresses only a target system generated by applying B refinements. However, MDE gives several solutions for defining model-to-code transformations which are often used to translate a given model into an executable implementation. We believe that a valuable application of code generation through a standard model transformation requires additional works in order to be able to reason on the equivalence between the generated code and the proofs done thanks to the DS(M)L's formal specifications.

References

1. Abrial JR (1996) The B-book: assigning programs to meanings. Cambridge University Press, New York
2. Bettini L (2013) Implementing domain-specific languages with Xtext and Xtend. Packt Publishing, Birmingham
3. Bousse E, Leroy D, Combemale B, Wimmer M, Baudry B (2018) Omniscient debugging for executable dsls. J Syst Softw 137:261–288. https://doi.org/10.1016/j.jss.2017.11.025
4. Bryant B, Gray J, Mernik M, Clarke P, France R, Karsai G (2011) Challenges and directions in formalizing the semantics of modeling languages. Comput Sci Inf Syst. https://doi.org/10.2298/CSIS110114012B
5. Combemale B, Brun C, Breathe life into your designer! Model simulation, animation and debugging with sirius animator, part of the GEMOC studio. http://gemoc.org/breathe-life-into-your-designer.html. Accessed 28 May 2020
6. Davies J, Milward D, Wang CW, Welch J (2015) Formal model-driven engineering of critical information systems. Sci Comput Program 103(C):88–113. https://doi.org/10.1016/j.scico.2014.11.004
7. Deantoni J (2016) Modeling the behavioral semantics of heterogeneous languages and their coordination. In: 2016 architecture-centric virtual integration (ACVI), pp 12–18
8. Dghaym D, Poppleton M, Snook C (2018) Diagram-led formal modelling using iUMLB for Hybrid ERTMS Level 3. In: ABZ 2018, vol 10817, pp 338–352. Springer. https://eprints.soton.ac.uk/417755/
9. Eclipse: Acceleo (2012). http://www.eclipse.org/acceleo/. Accessed 28 May 2020
10. Floyd RW (1993) Assigning meanings to programs. Springer, Dordrecht. https://doi.org/10.1007/978-94-011-1793-7_4
11. Gargantini A, Riccobene E, Scandurra P (2010) Combining formal methods and mde techniques for model-driven system design and analysis. Int J Adv Softw 1&2: 1–18
12. Group OM (2015) Meta Object Facility (MOF) 2.5.1 core specification. https://www.omg.org/spec/MOF/2.5.1/. Accessed 28 May 2020
13. Harel D, Rumpe B (2004) Meaningful modeling: what's the semantics of "semantics"? Computer 37:64–72. https://doi.org/10.1109/MC.2004.172
14. Hartmann T, Sadilek DA (2008) Undoing operational steps of domain-specific modeling languages. In: Proceedings of the 8th OOPSLA workshop on domain-specific modeling (DSM'08): University of Alabama at Birmingham
15. Idani A, Ledru Y (2015) B for modeling secure information systems: the b4msecure platform. In: International conference en formal engineering methods (ICFEM). LNCS, vol 9407. Springer, pp 312–318
16. Idani A, Ledru Y, Wakrime AA, Ayed RB, Bon P (2019) Towards a tool-based domain specific approach for railway systems modeling and validation. In: Third international conference on reliability, safety, and security of railway systems. LNCS, vol 11495. Springer, pp 23–40f
17. Idani A, Ledru Y, Wakrime AA, Ayed RB, Dutilleul SC (2019) Incremental development of a safety critical system combining formal methods and dsmls: application to a railway system. In: 24th international conference on formal methods for industrial critical systems. LNCS, vol 11687. Springer, pp 93–109
18. Idani A, Vega G, Leuschel M (2019) TTC'2019: the MEEDUSE solution. https://www.transformation-tool-contest.eu/papers/TTC19_paper_5.pdf. Accessed 28 May 2020
19. Jézéquel JM, Barais O, Fleurey F (2011) Model driven language engineering with kermeta. In: 3rd international summer school conference on generative and transformational techniques in software engineering III. GTTSE'09. Springer, Berlin, pp 201–221
20. Jézéquel J-M, Combemale B, Barais O, Monperrus M, Fouquet F (2015) Mashup of meta-languages and its implementation in the kermeta language workbench. Softw Syst Model 14(2):905–920
21. Jouault F, Allilaire F, Bézivin J, Kurtev I, Valduriez P (2006) Atl: a qvt-like transformation language. In: Companion to the 21st ACM SIGPLAN symposium on object-oriented programming systems. Languages, and applications, OOPSLA '06. ACM, New York, pp 719–720
22. Kim SK, Carrington D (1999) Formalizing the uml class diagram using object-z. In: Proceedings of the 2nd international conference on the unified modeling language: beyond the standard, UML'99, pp. 83–98. Springer, Berlin. http://dl.acm.org/citation.cfm?id=1767297.1767307

23. Langer P, Mayerhofer T, Kappel G (2014) Semantic model differencing utilizing behavioral semantics specifications. In: 17th international conference model-driven engineering languages and systems: MODELS. LNCS, vol 8767. Springer, pp 116–132
24. Lano K, Clark D, Androutsopoulos K (2004) Uml to b: formal verification of object-oriented models. In: Boiten EA, Derrick J, Smith G (eds) Integrated formal methods. Springer, Berlin, pp 187–206
25. Lausdahl K, Lintrup HKA, Larsen PG (2009) Connecting UML and VDM++ with open tool support. In: FM 2009: formal methods, second world congress, Eindhoven, The Netherlands, 2–6 Nov 2009. Proceedings, pp 563–578
26. Leuschel M, Butler M (2008) Prob: an automated analysis toolset for the b method. Int J Softw Tools Technol Transf 10(2):185–203. https://doi.org/10.1007/s10009-007-0063-9
27. Mammar A, Laleau R (2006) A formal approach based on UML and B for the specification and development of database applications. Autom Softw Eng 13(4):497–528. https://doi.org/10.1007/s10515-006-0273-5
28. Mayerhofer T, Langer P, Wimmer M, Kappel G (2013) Towards xmof: executable dsmls based on fuml. In: International conference on software language engineering: SLE. LNCS, vol 8225. Springer, pp 56–75
29. Merilinna J, Pärssinen J (2010) Verification and validation in the context of domain-specific modelling. In: Proceedings of the 10th workshop on domain-specific modeling. ACM, New York, pp 9:1–9:6
30. Muller PA, Fleurey F, Jézéquel JM (2005) Weaving executability into object-oriented meta-languages. In: Proceedings of MODELS/UML'2005. Montego Bay, Jamaica. https://hal.inria.fr/hal-00795095. Accessed 28 May 2020
31. OMG: OMG Object Constraint Language (OCL), Version 2.3.1. Object Management Group (2012). http://www.omg.org/spec/OCL/2.3.1/. Accessed 28 May 2020
32. Said MY, Butler M, Snook C (2015) A method of refinement in uml-b. Softw Syst Model 14(4):1557–1580. https://doi.org/10.1007/s10270-013-0391-z
33. Saldhana JA, Shatz SM (2000) Uml diagrams to object petri net models: an approach for modeling and analysis. In: International conference on software engineering and knowledge engineering, pp 103–110
34. Snook C, Butler M (2006) Uml-b: formal modeling and design aided by uml. ACM Trans Softw Eng Methodol 15(1):92–122
35. Snook C, Savicks V, Butler M (2012) Verification of UML models by translation to UML-B. In: Formal methods for components and objects. LNCS, vol 6957. Springer, pp 251–266
36. Soden M, Eichler H (2009) Towards a model execution framework for eclipse. In: Proceedings of the 1st workshop on behaviour modelling in model-driven architecture, BM-MDA '09. ACM, New York, pp 4:1–4:7
37. Stefanica A, Mihancea PF (2017) XCORE: support for developing program analysis tools. In: IEEE 24th international conference on software analysis, evolution and reengineering, SANER 2017, Klagenfurt, Austria, 20–24 Feb, pp 462–466 (2017). https://doi.org/10.1109/SANER.2017.7884654
38. Steinberg D, Budinsky F, Paternostro M, Merks E (2009) EMF: eclipse modeling framework 2.0, 2nd edn. Addison-Wesley Professional, Boston
39. Tikhonova U (2017) Engineering the dynamic semantics of domain specific languages. Ph.D. thesis, Department of Mathematics and Computer Science, Proefschrift
40. Tikhonova U (2017) Reusable specification templates for defining dynamic semantics of dsls. Softw Syst Model. https://doi.org/10.1007/s10270-017-0590-0
41. Tikhonova U, Manders M, Brand van den M, Andova S, Verhoeff T (2013) Applying model transformation and event-b for specifying an industrial dsl. In: Boulanger F, Famelis M, Ratiu D (eds) MoDeVVa 2013: workshop on Model Driven Engineering, Verification and Validation: proceedings of the 10th international workshop on model driven engineering, verification and validation co-located with 16th international conference on model driven engineering Miami, Florida, 1st Oct 2013, CEUR workshop proceedings, pp 41–50. CEUR-WS.org
42. Vergu V, Neron P, Visser E (2015) DynSem: a DSL for dynamic semantics specification. In: M. Fernández (ed) 26th international conference on rewriting techniques and applications (RTA 2015). Leibniz international proceedings in informatics (LIPIcs), vol 36, pp 365–378. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany. https://doi.org/10.4230/LIPIcs.RTA.2015.365. http://drops.dagstuhl.de/opus/volltexte/2015/5208
43. Wachsmuth G (2008) Modelling the operational semantics of domain-specific modelling languages. In: Lämmel R, Visser J, Saraiva J (eds) Generative and transformational techniques in software engineering II (GTTSE). Springer, Berlin, pp 506–520
44. Wildmoser M, Nipkow T (2004) Certifying machine code safety: shallow versus deep embedding. In: Slind K, Bunker A, Gopalakrishnan G (eds) Theorem proving in higher order logics (TPHOLs 2004). LNCS, vol 3223. Springer, pp 305–320
45. Zalila F, Crégut X, Pantel M (2013) Formal verification integration approach for dsml. In: Model-driven engineering languages and systems. Springer, pp 336–351