Applying Model Transformation and Event-B for Specifying an Industrial DSL

Ulyana Tikhonova, Maarten Manders, Mark van den Brand,
Suzana Andova, and Tom Verhoeff

Technische Universiteit Eindhoven,
P.O. Box 513, 5600 MB Eindhoven, The Netherlands
{u.tikhonova,m.w.manders,m.g.j.v.d.brand,s.andova,t.verhoeff}@tue.nl

Abstract. In this paper we describe our experience in applying the Event-B formalism for specifying the dynamic semantics of a real-life industrial DSL. The main objective of this work is to enable the industrial use of the broad spectrum of specification analysis tools that support Event-B. To leverage the usage of Event-B and its analysis techniques we developed model transformations, that allowed for automatic generation of Event-B specifications of the DSL programs. The model transformations implement a modular approach for specifying the semantics of the DSL and, therefore, improve scalability of the specifications and the reuse of their verification.

Key words: domain specific language, Event-B, model transformations, verification and validation, reuse, scalability

1 Introduction

Domain-Specific Languages (DSLs) are a central concept of Model Driven Engineering (MDE). A DSL provides domain notions and notation for defining models. It implements the semantic mapping of the models by means of model transformations. A DSL bridges the gap between the domain level and an execution platform. From a semantics point of view this gap can be quite wide, i.e. the DSL implementation usually includes rather complicated design solutions and algorithms. To manage the complexity of the industrial DSL, considered in this paper, we provide an explicit definition of its semantics by means of a formal method. This allows for formal specification of the DSL semantics and for assessing correctness of the specified semantic mapping via verification and validation.

In this paper we discuss the use cases of verification and validation applied to a DSL specification in an industrial context. We identify two different roles, that use different types of analysis of the DSL specification. A DSL developer is interested in validating and checking consistency of the DSL design and implementation. A DSL user is interested in getting better understanding of the DSL semantics, for example via simulation of its specifications. Correspondingly, in

the context of MDE a formal specification of a DSL can be given on two abstraction levels: the DSL metamodel level and the DSL model level.

There exists quite a number of formalisms for specifying behavior and tools for analyzing these specifications using different verification and validation techniques. In this research we use the Event-B formalism [2] and the Rodin platform [3], as they allow the implementation of all use cases listed above. By using Event-B and Rodin we can (1) prove consistency of the DSL semantics specifications with (automatic and interactive) provers, (2) find deadlocks and termination problems using model checkers, (3) use animators to validate the specified semantics with the help of domain experts, and (4) provide graphical visualization of the specification to help DSL users to understand how their programs run. All these tools are available in Rodin for Event-B.

In this paper, we show how Event-B and Rodin can be adopted in practice and applied to the industrial use cases, through model transformations from the DSL to Event-B. Our model transformations can automatically generate an Event-B specification for each concrete DSL program. For this, we apply the techniques of *composition* and *instantiation* of Event-B specifications. Composition of Event-B specifications simplifies the creation, maintenance and verification of larger specifications, because one can handle the smaller components separately. Instantiation is a way to concretize a generic Event-B specification, defined on the DSL metamodel level, to the model level of a concrete DSL program. The instantiation technique allows for the reuse of verification results from one level to the other. As a result of applying these techniques, our model transformations improve usability of Event-B and Rodin.

In the rest of the paper, Section 2 gives the overview of the industrial DSL and defines roles and use cases; Section 3 introduces the Event-B formalism; Sections 4.1 and 4.2 describe the decomposition and instantiation techniques; Section 4.3 outlines the implementation of our approach and results of its application. Related work is discussed in Section 5. Conclusions and directions for future work are given in Section 6.

2 Case Study and Use Cases

Our case study was performed at ASML,[1] a producer of complex lithography machines for the semiconductor industry. In our case study we specified the dynamic semantics of the LACE DSL. LACE (Logical Action Component Environment) is one of the DSLs, developed by and used within ASML for controlling lithography machines. A lithography machine consists of many physical *subsystems* (such as actuators, projectors, and sensors), which operate simultaneously in order to perform the required functions of the machine. LACE allows for specifying how subsystems operate in collaboration with each other by means of so-called *logical actions*. An example of a logical action is shown in Figure 1.

LACE has a graphical notation based on UML activity diagrams. A logical action consists of *subsystem actions* (rounded rectangles in Figure 1), each of

[1] `www.asml.com`, `http://en.wikipedia.org/wiki/ASML_Holding`

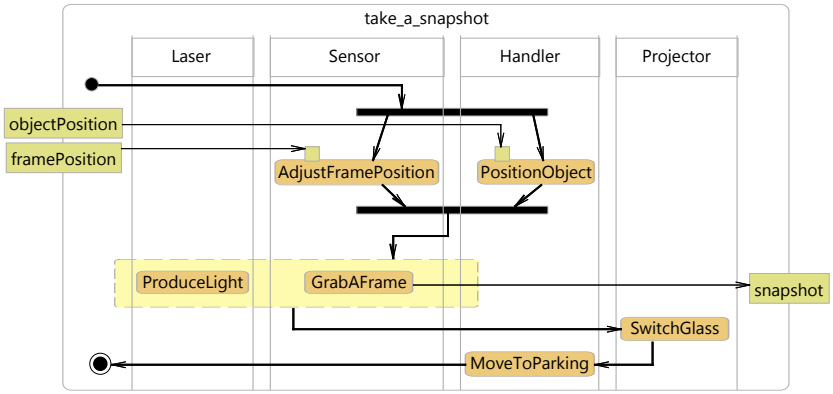

Fig. 1: An example of a logical action for taking a snapshot

which belongs to a subsystem, that executes this subsystem action (vertical column, containing the rounded rectangle). Subsystem actions, combined together into a so-called *scan* (dashed rounded rectangle), are executed synchronously. Thus, in Figure 1 the `Sensor` subsystem starts and stops the `GrabAFrame` action at the same moments, as the `Laser` subsystem starts and stops the `ProduceLight` action. Subsystem actions within a logical action can be executed *sequentially or concurrently* (thick arrows, fork and join nodes). For example, the subsystem actions `AdjustFramePosition` and `PositionObject` are independent actions and can be executed in any order or in parallel, but the `GrabAFrame` action can be performed only after both these actions are finished. Finally, subsystem actions may require and produce *data*. The dataflow in a logical action is depicted by means of thin arrows, input and output pins. For example, in Figure 1 the `GrabAFrame` action produces data, which is saved in the `snapshot` output parameter.

The high-level description of the machine subsystems' behavior, given in logical actions, is translated into the invocations of hardware drivers and a synchronization driver in such a way that the resulting execution matches the behavior specified in the logical actions. The semantic gap between LACE and driver functions is wide, and thus the translation is hard to develop, maintain, understand, and use. We construct a formal specification of LACE and apply different kinds of analysis to it, in order to enhance understandability, maintainability and usage of the DSL translation. We identify the following roles and use cases of specification of a DSL and its analysis in the industrial context (Figure 2).

A *DSL developer* designs and develops the DSL by constructing its metamodel and semantic translations. Formal specification of this DSL implementation allows for the verification, such as checking that it is consistent, non-contradictive, feasible and complete. A *DSL user* specifies DSL programs as instances of the DSL metamodel. In the context of formal methods the instantiation of metamodel by a DSL program needs to be verified separately. Formal specification of the DSL programs allows for the execution of specifications, and

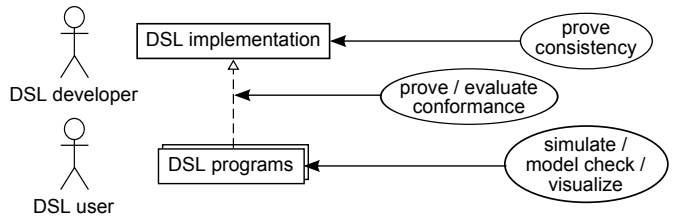

Fig. 2: Roles and use cases of the DSL specification and its analysis

thus for model checking and simulation. The construction of the LACE specifications and the implementation of the listed use cases are described in Section 4.

3 Event-B

Event-B is an evolution of the B method, both introduced by Abrial [2]. Event-B employs set theory and first-order logic for specifying software and/or hardware behavior. A big advantage of Event-B is its tool support, offered by the Rodin platform [3]. Using Rodin and its plug-ins, one can create and edit Event-B specifications, verify them using automatic or interactive provers, animate and model check Event-B specifications.

An Event-B specification consists of *contexts* and *machines*. A context describes the static part of a system: *sets*, *constants* and *axioms*. A machine uses (*sees*) the context to specify behavior of a system via a state-based formalism. *Variables* of the machine define the state space. *Events*, which change values of these variables, define transitions between the states. An event consists of *guards* and *actions*, and can have *parameters*. An event can occur only when its guards are true, and as a result of the event its actions are executed. The properties of the system are specified as *invariants*, which should hold for all reachable states. The properties are verified via proving automatically generated *proof obligations* and/or via model checking.

The attractive simplicity of Event-B is enhanced by techniques such as *shared event composition* and *generic instantiation*, which support scalability and reuse of Event-B specifications [4]. We discuss these techniques in detail in Section 4.

4 Model Transformations from LACE to Event-B

The Rodin platform allows for the implementation of the use cases described in Figure 2, provided that the corresponding Event-B specifications of LACE are given. This poses the following problems. First, the semantics of LACE is complex, therefore capturing it within Event-B machines is challenging and results in a big specification, which is hard to understand, maintain and verify. To tackle this problem we apply two types of composition of Event-B specifications: *composition of semantic features* and *composition of machines* (Section 4.2). The

LACE specification is composed using model transformations. Second, while a specification of LACE on the metamodel level can be created and analyzed once, specifications of the LACE programs need to be constructed and analyzed many times by DSL users. We cannot expect DSL users to create Event-B specifications of their LACE programs and to verify them themselves. Therefore, we apply model transformations from LACE to Event-B to automatically generate specifications of LACE programs and we use the *generic instantiation* technique to verify their conformance to the LACE specification, given on the metamodel level (Section 4.1). Moreover, we enhance simulation of Event-B specifications of LACE programs by providing a user-friendly visualization.

4.1 Instantiation of Event-B specification

Generic instantiation is a technique, proposed by Abrial and Hallerstede to reuse an existing Event-B specification by refining the data structures, specified in its constants and variables, in a new copy of this specification [4]. We apply generic instantiation as depicted in Figure 3 (on the left). The concepts of `conceptual machine` and of `composite machine` are introduced in Subsection 4.2.

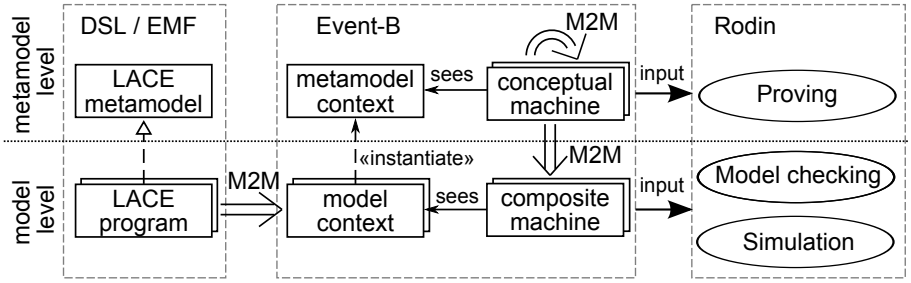

Fig. 3: Instantiation and composition of Event-B specification

The `metamodel context` captures the structure specified in the LACE metamodel. A `conceptual machine` uses this context to specify the dynamic semantics of LACE in terms of the metamodel. Based on the structural properties, specified in the axioms of the `metamodel context`, the `conceptual machine`s are proved to be consistent and complete by discharging the corresponding proof obligations using the Rodin provers. Thus, the semantics is verified on the metamodel level. The `metamodel context` and the `conceptual machine`s for a specific DSL are constructed manually and only once.

In the `model context`, values are assigned to the sets and constants, introduced in the `metamodel context`. The assignments are done in the axioms of the `model context`. Therefore, being used by a `composite machine`, this context specifies behavior of a concrete LACE program – on the model level. This specification can be model checked and animated, allowing for the analysis of a

particular LACE program. `Model contexts` are generated from LACE programs automatically by means of model transformations.

According to the generic instantiation technique, if all structural properties, defined in the `metamodel context`, can be derived for the structure, instantiated in the `model context`, then the verification of the Event-B specification can be extended straightforwardly from the metamodel level to the model level [4]. In [13] and [6] it is proposed to use theorem proving to show this derivation.

Due to the large sizes of the `model contexts`, generated from LACE programs, the automatic provers of Rodin fail to discharge instantiation theorems. On the other hand, we do not expect an average DSL user to prove these theorems using Rodin interactive provers, as it requires knowledge of propositional calculus and understanding of proof strategies. Therefore, instead of the theorem proving, we employ evaluation of structural properties predicates in the ProB animator integrated in Rodin [11]. Thus, we achieve automatic proof of instantiation in Event-B.

4.2 Composition of Event-B specification

As we mentioned before, capturing semantics of LACE within Event-B machines is rather complicated due to their different abstraction levels. To handle this complexity we employ modularity of LACE semantics. Each module is described separately as a *conceptual machine* in Event-B. Composition of the conceptual machines gives a resulting Event-B machine, which specifies the LACE dynamic semantics. The modular approach facilitates development, understandability and proving the correctness of the specification. We distinguish two types of modularity in the dynamic semantics of LACE: *semantic modularity* and *architectural modularity*.

To manage complexity of the LACE semantics we decompose it into separate *semantic features* (SFs): `Core SF`, `Order SF`, `Scan SF` and `Data SF`. The `Core SF` specification defines common concepts and interfaces: logical actions, consisting of subsystem actions, subsystems and events for requesting execution of logical actions and subsystem actions. `Order SF`, `Scan SF` and `Dataflow SF` are specified independently on the basis of the `Core SF` machine by adding extra variables, invariants, parameters, guards and actions to the `Core SF` machine and by changing some of the Event-B types, used in it. `Order SF` introduces partial order of execution of subsystem actions within a logical action. `Scan SF` joins subsystem actions into scans. `Data SF` introduces input and output parameters and dataflow within a logical action. The composition of semantic features is implemented via weaving Event-B code of the machines in the model transformation (the self-referential `M2M` arrow in Figure 3).

The LACE implementation consists of different software components, such as: logical action components (`LAC`), that translate logical action requests into subsystem actions, and subsystems (`SS`), that execute subsystem actions. This architectural modularity of LACE is implemented in Event-B using the *shared event composition* approach [14]. Software modules are specified in separate machines, which are then composed into one `composite machine` specifying the

whole system. The interaction of the modules is implemented via composition (or in fact, synchronization) of the events of the composing machines. Composition of events means conjunction of the events' guards and composition of the events' actions in one composite event. The composition of the `LAC` and `SS` machines is implemented using model transformation (the `M2M` arrow from `conceptual machine`s to `composite machine`s in Figure 3).

As a result of the intersection of two types of specification modularity, eight conceptual machines and four composition schemes need to be specified: for each semantic feature we specify a conceptual machine of each software module (`LAC` and `SS`) and a scheme of the interaction of `LAC` with `SS`. An Event-B machine that specifies the LACE semantics as a whole is composed of `LAC` and `SS` machines, that include Event-B code for all four semantic features, according to the compositional schemes of all four semantic features. Two dimensions of the modularity presented above simplify creation, verification and validation of Event-B components and maintenance of the model transformations.

4.3 Implementation

The LACE-to-Event-B transformations, described in Sections 4.1 and 4.2, were implemented using the *Operational QVT* (Query/View/Transformation) language [1] in the Eclipse environment. The input for the transformation is provided directly by the LACE implementation software, which employs model transformation and code generation techniques in the Borland Together environment, and therefore is compatible with EMF (Eclipse Modeling Framework). As a target metamodel for the transformation we use the Event-B Ecore implementation provided by the EMF framework for Event-B [15].

The LACE-to-EventB transformation is designed in a modular way, which follows the logic of instantiation and composition techniques as described in Sections 4.1 and 4.2. Thus, the transformation is possible to reuse and to generalize.

Table 1: Characteristics of the LACE-to-Event-B transformation

Event-B components	Semantic features				core+scan+order+data
	core	scan	order	data	
Metamodel context	3 constants 5 axioms	4 constants 8 axioms	5 constants 9 axioms	11 constants 22 axioms	–
Model context	20 constants 7 axioms	21 constants 8 axioms	23 constants 10 axioms	37 constants 17 axioms	–
LAC machine	3 events 21 POs	3 events 23 POs	4 events 26 POs	4 events 28 POs	4 events 34 POs
SS machine	3 events 7 POs	3 events 11 POs	3 events 7 POs	3 events 7 POs	3 events 11 POs
composition of LAC and SS machines	10 events 70 POs	30 events 386 POs	10 events 82 POs	10 events 89 POs	30 events 491 POs

Table 1 shows the representative characteristics of the transformation: sizes of the metamodel contexts vs. model contexts and sizes of the conceptual machines (`LAC` and `SS` machines for `core`, `scan`, `order` and `data` semantic features) vs. composite machines (bottom row). The automatically generated Event-B components are shaded. As an input for the transformation the LACE program depicted in Figure 1 is used. All proof obligations (POs) of the LAC and SS machines are discharged by invocation of the automatic provers in Rodin. The proof obligations of the composite machines (bottom row) can be left undischarged, as these are inherited proof obligations of the LAC and SS machines (according to the shared event composition approach [14]). The Event-B machine that specifies the LACE semantics as a whole is located in the bottom right cell of the table. One can observe that this machine is much larger and has much more proof obligations, than the conceptual machines, of which this machine is composed.

To make it convenient for a LACE user to work with Event-B we developed a graphical visualization of the LACE specification using the BMotion Studio plug-in [10]. This visualization runs together with the ProB animator and provides a GUI (graphical user interface) for a machine being animated. The GUI is based on the original LACE notation. By experimenting with a LACE program specification using this GUI a user can get better understanding of the DSL design and improve efficiency of her programs. Screen shots of the visualization can be found on the web-page of our project.[2]

5 Related Work

There are a number of studies in which Event-B has been applied to a specification of the dynamic semantics of a DSL. Aït-Sadoune and Aït-Ameur employ Event-B and Rodin for proving properties and animation of BPEL processes [5]. Hoang et al. use Event-B and Rodin to automate analysis of Shadow models [9]. In both studies, DSL program descriptions are translated into Event-B specifications. The translations are implemented in the Java programming language. These works do not use generic instantiation and composition techniques, but apply *refinement* of Event-B machines [4] to implement modularity of the programs. Based on our experience, refinement restricts semantics definition and can be rather complicated for automatic proving. Moreover, we use model transformations to implement the generation of Event-B specifications, which increases the abstraction level of the translation and therefore enhances its reuse.

Besides Event-B, other formalisms have been used as a target formal domain for specifying semantics of DSLs. Chen et al. propose transformational specification of dynamic semantics using Abstract State Machines (ASM) as a target formalism and explore specified behavior by means of the AsmL simulator tool [8]. Moreover, they introduce *semantic units* as an intermediate common language for defining dynamic semantics of DSLs and explore a technique for their composition [7]. Another approach, that supports reuse of the DSL analyses via intermediate specification modules, is proposed by Ratiu et al. [12]. They

[2] `www.win.tue.nl/mdse/COREF`

identify conceptually distinct *sub-languages*, shared by different DSLs, and transform these to different analysis formalisms. These works support modularity of a DSL specification by modularizing target formalisms. In this paper we describe how modularity of the DSL specification arises from the modularity of the DSL semantics, and apply model transformations to compose semantic modules.

6 Conclusion

In this paper we showed how the dynamic semantics of an industrial DSL can be defined using the Event-B formalism and model transformations. The Rodin platform and its plug-ins provide a broad spectrum of functionality and analysis tools for Event-B specifications. Our objective was to adopt Event-B for the industrial use cases for two major roles: DSL users and DSL developers. This was achieved by using MDE techniques – model transformations that define the semantics mapping from DSL domain to Event-B.

In order to specify semantics of LACE in a modular and scalable way we introduce semantic features and specify them in conceptual Event-B machines. The conceptual machines are verified on the metamodel level using automatic provers of Rodin. The LACE-to-Event-B transformation composes the conceptual machines into the LACE specification and instantiates this specification for concrete LACE programs. The resulting Event-B specifications can be validated and model checked by DSL developers and can be simulated by DSL users in the user-friendly GUI – all in the Rodin environment.

As future work we aim for applying the demonstrated techniques to other DSLs. For this we need to generalize LACE-to-Event-B transformation by distinguishing repetitive Event-B code, that can be combined into fine-grained specification patterns. This will allow not only for reuse of the demonstrated techniques of instantiation and composition, but also for reuse of already verified and visualized pieces of specification.

7 Acknowledgements

We are very grateful to Marc Hamilton and Wilbert Alberts (ASML, The Netherlands) for introducing us to the LACE world and providing very useful feedback on our experiments. We would like to thank Michael Butler and Colin Snook (University of Southampton, United Kingdom) for their help with using Event-B and Rodin. We also would like to thank Anton Wijs and Alexander Serebrenik (Eindhoven University of Technology, The Netherlands) for their useful comments on this paper.

References

1. *Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification.* Object Management Group (OMG), July 2007.

2. J.-R. Abrial. *Modeling in Event-B: system and software engineering*, volume 1. Cambridge University Press, 2010.
3. J.-R. Abrial, M. Butler, S. Hallerstede, T. S. Hoang, F. Mehta, and L. Voisin. Rodin: An Open Toolset for Modelling and Reasoning in Event-B. *International Journal on Software Tools for Technology Transfer (STTT)*, 12(6):447–466, 2010.
4. J.-R. Abrial and S. Hallerstede. Refinement, Decomposition, and Instantiation of Discrete Models: Application to Event-B. *Fundam. Inform.*, 77(1-2):1–28, 2007.
5. I. Aït-Sadoune and Y. Aït-Ameur. Stepwise Design of BPEL Web Services Compositions: An Event-B Refinement Based Approach. In R. Lee, O. Ormandjieva, A. Abran, and C. Constantinides, editors, *Software Engineering Research, Management and Applications*, pages 51–68. Springer Berlin / Heidelberg, 2010.
6. D. A. Basin, A. Fürst, T. S. Hoang, K. Miyazaki, and N. Sato. Abstract Data Types in Event-B – An Application of Generic Instantiation. In *Workshop on the experience of and advances in developing dependable systems in Event-B*, volume abs/1210.7283 of *CoRR*, pages 5–16, 2012.
7. K. Chen, J. Porter, J. Sztipanovits, and S. Neema. Compositional Specification Of Behavioral Semantics For Domain-Specific Modeling Languages. *Int. J. Semantic Computing*, 3:31–56, 2009.
8. K. Chen, J. Sztipanovits, S. Abdelwalhed, and E. Jackson. Semantic Anchoring with Model Transformations. *European Conference on Model Driven Architecture - Foundations and Applications*, pages 115–129, 2005.
9. T. S. Hoang, A. McIver, L. Meinicke, C. Morgan, A. Sloane, and E. Susatyo. Abstractions of non-interference security: probabilistic versus possibilistic. *Formal Aspects of Computing*, pages 1–26, 2012.
10. L. Ladenberger, J. Bendisposto, and M. Leuschel. Visualising Event-B Models with B-Motion Studio. In M. Alpuente, B. Cook, and C. Joubert, editors, *Proceedings of FMICS 2009*, volume 5825 of *Lecture Notes in Computer Science*, pages 202–204. Springer, 2009.
11. M. Leuschel and M. Butler. ProB: A Model Checker for B. In A. Keijiro, S. Gnesi, and M. Dino, editors, *FME*, volume 2805 of *Lecture Notes in Computer Science*, pages 855–874. Springer-Verlag, 2003.
12. D. Ratiu, M. Voelter, Z. Molotnikov, and B. Schaetz. Implementing Modular Domain Specific Languages and Analyses. In *Proceedings of the Workshop on Model-Driven Engineering, Verification and Validation*, pages 35–40, 2012.
13. R. Silva and M. Butler. Supporting Reuse of Event-B Developments through Generic Instantiation. In K. Breitman and A. Cavalcanti, editors, *11th International Conference on Formal Engineering Methods (ICFEM)*, volume 5885 of *Lecture Notes in Computer Science*, pages 466–484. Springer, 2009.
14. R. Silva and M. Butler. Shared Event Composition/Decomposition in Event-B. In B. K. Aichernig, F. S. de Boer, and M. M. Bonsangue, editors, *Formal Methods for Components and Objects (FMCO)*, pages 122–141. Springer, 2010.
15. C. Snook, F. Fritz, and A. Illisaov. An EMF Framework for Event-B. In *Workshop on Tool Building in Formal Methods - ABZ Conference*, 2010.