Visualization of Formal Specifications for Understanding and Debugging an Industrial DSL

Ulyana Tikhonova(✉), Maarten Manders, and Rimco Boudewijns

Technische Universiteit Eindhoven,
P.O. Box 513, 5600 MB Eindhoven, The Netherlands
{u.tikhonova,m.w.manders}@tue.nl, r.c.boudewijns@alumnus.tue.nl

Abstract. In this work we report on our proof of concept of a generic approach: visualized formal specification of a Domain Specific Language (DSL) can be used for debugging, understanding, and impact analysis of the DSL programs. In our case study we provide a domain-specific visualization for the Event-B specification of a real-life industrial DSL and perform a user study among DSL engineers to discover opportunities for its application. In this paper, we explain the rationale behind our visualization design, discuss the technical challenges of its realization and how these challenges were solved using the Model Driven Engineering (MDE) techniques. Based on the positive feedback of the user study, we present our vision on how this successful experience can be reused and the approach can be generalized for other DSLs.

Keywords: Event-B · Visualization · Domain specific language · User study

1 Introduction and Motivation

Domain-Specific Languages (DSLs) are a central concept of Model Driven Engineering (MDE). A DSL is a programming language specialized to a specific application domain. It captures domain knowledge and supports the reuse of such knowledge via common domain notions and notation. In this way, the DSL raises the abstraction level for solving problems in the domain. DSLs are considered to be very effective in software development and are being widely adopted by industry nowadays.

DSL semantics is usually implemented as a translation from the domain concepts to a programming language for an execution platform. From a semantics point of view, the gap bridged by this translation can be quite wide. The DSL translation usually includes complex design solutions and algorithms, which employ both high-level concepts of the DSL and low-level concepts of the execution platform. In traditional MDE, all details of the DSL translation reside in model transformations and code generators. So, to understand how a DSL program works one can either examine the source code of the DSL translation,

P. Milazzo et al. (Eds.): STAF 2016, LNCS 9946, pp. 179–195, 2016.
DOI: 10.1007/978-3-319-50230-4_13

or the source code generated from the DSL program. This poses undesirable challenges when developing, understanding, and debugging DSL programs.

A thorough mathematical-based *formal specification* of a DSL can make the DSL translation more accessible, by expressing the translation on a higher abstraction level than model transformations and code generators [14]. Moreover, tools supporting this formal specification can enhance understanding, developing, and debugging of DSL programs. For example, DSL programs can be explored and debugged by execution (simulation) of their specification using animation tools. However, practical realization of these benefits requires adaptation of the formal specification and supporting tools to the needs and background of engineers who develop or use the DSL.

In this work, we provide a domain-specific visualization for the formal specification of a real-life industrial DSL. This visualization mimics the original graphical notation of the DSL and runs on top of the animation of the DSL specification. As a result, the DSL specification can be used by DSL engineers who are not familiar with the formal notation of the specification. We implemented this visualization by means of model transformations from the DSL to the visualization platform. Thus, a visualization is generated automatically for each concrete DSL program. Furthermore, we investigated the needs and perception of DSL engineers by means of a user study performed with this visualization. The user study indicated that there is a need for such visualization of DSL models; and that it might be challenging to keep the DSL specification consistent with the actual implementation of the DSL, which evolves over time. This challenge is addressed in a framework that we propose as our vision on how successful specification solutions and their visualizations can be reused for other DSLs in the form of specification and visualization templates.

2 Related Work

There exist various approaches for visualizing a formal specification and its execution (animation). It can be a visualization of a state space, or wrapping a formalism into the (standard) graphical notation of UML (such as [10]). In our work we focus on a domain-specific visualization – on a graphical representation tailored for a specific (engineering or application) domain. A number of case studies have proven that a domain-specific visualization of a formal specification can be very useful for creating, validating, and applying the specification by humans, especially by domain experts. For example, in [4] Hansen et al. state that graphical representation of their Event-B specification of a landing gear system was crucial for its development and validation. In [7] Mathijssen and Pretorius use a visualization of their mCRL2 specification of an automated parking garage to discover and fix a number of bugs in the specification. They conclude that the visualized simulation is more intuitively clear and easy to understand, and thus, helps to identify issues that may not have been noted otherwise.

In [12] Stappers specifies the behavior of an industrial wafer handler using mCRL2, obtains a trace to a deadlock state using the mCRL2 toolset, and visualizes this trace using a CAD (Computer Aided Design) model of the wafer

handler in a kinematic 3D visualizer. In other words, the visualization animates the 3D virtual model of the physical system by moving its parts along the predefined motion paths. As a result, engineers of the wafer handler could identify the problem that leads to the deadlock state and find a proper solution. Stappers presents his approach from a general point of view, describing the components that are required to realize such a visualization and their architecture. For our work we draw inspiration from his motivation on how system development can benefit from formal specifications and their visualization, and from his overview of how various technological fields connect and interact with each other in order to implement these benefits.

While most of domain-specific visualizations are implemented in an ad-hoc way (for example, a traffic light system presented in [8] is visualized in a prototype simulator developed in Java specifically for this case study); recent developments facilitate the creation of visualizations using dedicated graphical editors integrated with a formalism toolset. For instance, in BMotion Studio [5] (integrated with the ProB animator[1]) one can create an interactive domain-specific visualization for a (single) B or Event-B specification. BMotion Studio has been successfully applied to a number of case studies (see for example [4,6]). In our work we lift this successful tool support to the level of DSLs, automating the creation of BMotion Studio visualizations for multiple (or for a family of) Event-B specifications.

To date, there is a lack of tool support for understanding and debugging an executable DSL on the level of its domain rather than on the level of its target execution platform (such as generated C or Java code). In [3] Chis et al. recognize this problem and propose the Moldable Debugger framework for developing domain-specific debuggers. The Moldable Debugger allows for configuring a domain-specific debugging view as a collection of graphical widgets, such as stack, code editor, and object inspector. In order to visualize the execution of a DSL program in such widgets, a DSL developer needs to specify so-called debugging predicates (capturing an execution state) and debugging operations (controlling the execution of a program). To realize this approach, the Moldable Debugger builds on top of and extends an existing IDE (integrated development environment). In our work we build on top of formal methods, making use of the DSL formal specification. Moreover, we discuss how to design the visualization of a DSL program (*i.e.* domain-specific graphical widgets) based on an explicit definition of the DSL dynamic semantics.

In [2] Bandener et al. visualize behavior of a graphical language in the form of the animated concrete syntax using the Dynamic Meta Modeling (DMM) technique. In DMM, a so-called runtime metamodel enhances the language metamodel with concepts that express an execution state. The language behavior (its dynamic semantics) is specified as a set of graph transformation rules for deriving instances of the runtime metamodel. When applied to a program (*i.e.* instance of the language metamodel), these rules iteratively generate a state space representing the behavior of the given program. Each of these states is

[1] http://www3.hhu.de/stups/prob/.

an instance of the runtime metamodel. Bandener et al. enhance the language concrete syntax with the graphical representation of the runtime metamodel. As a result, a graphical representation (*i.e.* a diagram) can be generated for each state of the state space. Their front-end tool allows for choosing a path in such a visualized state space. In our work we also strive for the effect of the animated concrete syntax of the DSL.

3 Visualization of DSL Specifications

In our previous work [13], we employed the Event-B formalism for specifying the dynamic semantics of an industrial DSL. The main motivation for choosing Event-B rather than a formalism designed for specifying dynamic semantics of programming languages (such as Action Semantics or Structural Operational Semantics), was that the Rodin platform offers various supporting tools for Event-B: editors, generator of proof obligations, automatic provers, animators, etc. To be able to apply these tools to a DSL specification, we use Event-B as a back-end formalism for defining the DSL dynamic semantics and develop model-to-model transformations from the DSL to Event-B.

In this work, we apply one of the Rodin tools to Event-B specifications of the DSL. Namely, we employ BMotion Studio for visualizing animation (*i.e.* execution of specifications) of various DSL programs. In this way we help DSL engineers to understand how their programs are executed, *i.e.* we realize specification-based domain-specific debugging. For this, we automate the construction of domain specific visualizations of Event-B specifications using model-to-model transformations from the DSL to BMotion Studio. In what follows, we give an overview of our case study and of our approach to specify the dynamic semantics of the DSL in Event-B (Sect. 3.1), and discuss how we design and implement a visualization of DSL specifications (Sects. 3.2 and 3.3).

3.1 Specification of the LACE DSL

LACE (Logical Action Component Environment) is a mature industrial DSL, developed by and used within ASML[2], a world leading company that produces wafer steppers. LACE is used for (automatic) generation of software that controls ASML lithography machines and orchestrates their numerous subsystems by invoking drivers in a synchronized and effective way. LACE has a graphical notation based on UML activity diagrams. An example of a LACE program is depicted in Fig. 1(a). Here, each column represents a subsystem driver, the rectangles in the columns represent actions of these subsystems, and arrows represent the control (thick arrows) and data (thin arrows) flow. A LACE diagram is translated into C-code, which is executed on a target machine (wafer stepper). Such a translation bridges the wide semantic gap between high-level concepts of the DSL and low-level concepts of the execution platform. Moreover, it includes

[2] www.asml.com.

rather complicated design solutions and algorithms. We elicited these design solutions and algorithms by specifying them using the Event-B formalism [1]. The resulting formal specification reveals the complexity of the DSL translation and, thus, facilitates its management.

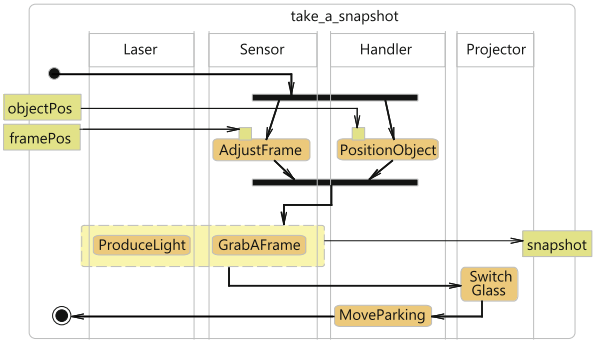
In Event-B a system is specified as a set of *variables*, that define the state space, and a set of *events*, that define transitions between the states. While in general such a formalism allows for specifying a system on any level of abstraction (from requirements to an implementation), in practice, applying Event-B to the DSL semantics results in a big specification, which is hard to understand, maintain, and verify. There exist a number of techniques that tackle this problem by building on top of Event-B and allowing for (de)composition of an Event-B specification into/of separate Event-B specifications. We apply shared event (de)composition [9] to modularize the specification of LACE.

In the MDE context a DSL resides in two abstraction levels: the DSL metamodel and DSL models (programs). The DSL is designed and implemented on the metamodel level, and it is used via instantiating DSL programs on the model level. While a generic specification of the DSL on the metamodel level can be created and analyzed once, Event-B specifications of many DSL programs need to be constructed for and simulated by DSL users (engineers, who program in LACE). We cannot expect DSL users to create Event-B specifications of their DSL programs and to use Event-B tools themselves. Therefore, we generate such Event-B specifications automatically by instantiating the LACE specification for each concrete LACE program.

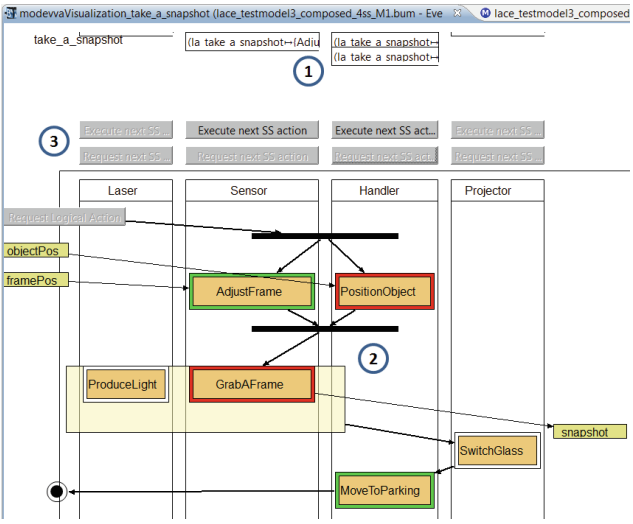
Thus, on the meta-model level, we define LACE as a set of separate Event-B *meta-specifications* of different aspects of the semantics. For example, the buffered execution on a subsystem driver is represented and specified as a queue; the mutual dependency of driver operations (such as `AdjustFrame` and `GrabA-Frame` in Fig. 1(a)) is represented and specified as a partial order. On the model level, our DSL-to-Event-B model transformation instantiates and composes such meta-specifications together using an input DSL program as a configuration instruction. For example, for the LACE program depicted in Fig. 1(a) we generate four instances of the queue meta-specification (corresponding to the four subsystems `Laser`, `Sensor`, `Handler`, and `Projector`, as they appear in Fig. 1(a)) and one instance of the partial order meta-specification; and compose these five Event-B specifications together. As a result, an Event-B specification is generated automatically for each concrete DSL program, and the DSL semantics is specified in a clear and modular way.

3.2 Visualization of the LACE DSL

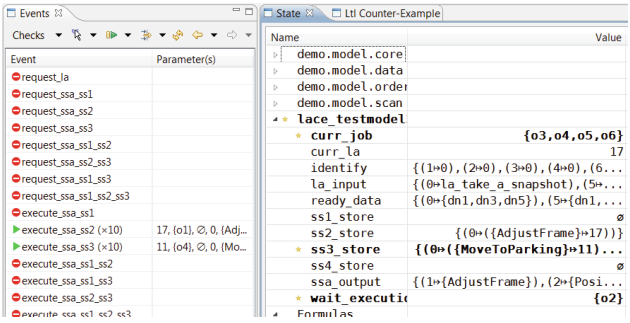
In order to understand how LACE programs are executed, explore the LACE semantics, and even debug the programs, we simulate LACE programs using their Event-B specifications. The simulation of a LACE program is achieved through the execution of the Event-B specification (generated for this program) in an animation tool, for example in the ProB animator. However, DSL engineers



(a) Original DSL program



(b) Visualized animation in ProB: (1) subsystem queues, (2) highlighted actions, (3) buttons for executing events



(c) Animation in ProB (running underneath the visualization)

Fig. 1. An example of the LACE DSL program and its visualized animation in ProB (Color figure online)

are hardly familiar with the formal notation of Event-B employed by the animation tools. Moreover, they are hardly familiar with the semantics of the DSL as it is specified in Event-B, and thus, cannot connect Event-B specifications generated from their programs with their original programs.

For example, Fig. 1(c) shows a fragment of the screen shot of how the Event-B specification, which is generated from the LACE program depicted in Fig. 1(a), is executed in the ProB animator. This Event-B specification is composed of the four instances of the same meta-specification representing a subsystem driver (for `Laser`, `Sensor`, `Handler`, and `Projector`) and of one instance of the partial order meta-specification. The events of these instantiated specifications and, moreover, their combinations can be seen on the left side of Fig. 1(c) (tab "Events"). The variables of the instantiated specifications and their values can be seen on the right side of Fig. 1(c) (tab "State"). It is hard to trace these events and variables back to the original LACE program without knowing the meta-specifications and how they are instantiated and composed.

Consequently, while such a simulation might be useful for a designer who creates Event-B specifications of the DSL; one cannot expect engineers, who program using the DSL notation, to be able to use such a simulation. Therefore, we create a domain-specific visualization for Event-B specifications of LACE. For this we employ BMotion Studio, that provides a graphical editor for creating such visualizations and uses Event-B notation for specifying various details, such as predicate and value expressions.[3] As BMotion Studio is integrated into ProB, a visualization runs together with (or on top of) the ProB animator.

A visualization provides a graphical user interface (GUI) for animating an Event-B specification. For creating such a GUI, BMotion Studio offers a palette of various graphical elements (shapes, connectors, text fields, buttons, etc.) and two concepts responsible for the animation: *controls* and *observers*. Controls are used to execute events of the specification. Observers are used to visualize the current state (*i.e.* current values of the variables). Below we discuss how such a GUI can be designed for a DSL.

The main goal of our DSL visualization is to help a DSL user to grasp how a DSL program is executed. For this, we start from what a user already knows: the DSL concrete syntax. We add to it what a user aims to discover: the DSL dynamic semantics. In other words, we base our visualization on the graphical syntax of LACE; and we project the concepts of the LACE dynamic semantics on it. In general, such concepts represent an execution state and operations that can change this execution state. However, it is not obvious how such concepts can be visualized on top of the graphical syntax of LACE.

For textual programming languages, such as C or Java, an established way to visualize an execution state on the top of the program text is to highlight the code line that is being executed. Moreover, additional views, such as call stack, memory, variables, etc. allow for inspecting other aspects of the execution state. The execution of a program is done step by step using various play-buttons.

[3] In this work we use the first version of BMotion Studio. Currently the new version, BMotionWeb, is available.

This way of visualizing a program execution directly links to the concepts that are usually used in descriptions of the dynamic semantics of such languages: execution of a program line-by-line, function calls, memory management.

An example of a graphical language is considered in [2]. In this work the execution of a UML activity diagram is visualized by projecting its dynamic semantics on the diagram. As the dynamic semantics of UML activity diagrams is described using the concept of token (adopted from Petri nets), the visualization consists of tokens (filled black circles) moving between activity nodes.

The dynamic semantics of LACE is defined as a set of meta-specifications (as described in Sect. 3.1).[4] Therefore, we visualize concepts defined in these meta-specifications and project them on the LACE concrete syntax. This allows for breaking down the visualization design and striving towards a modular solution, by focusing on the visualization of meta-specifications rather than on the visualization of the dynamic semantics as a whole. Below we demonstrate this approach on the examples of two meta-specifications (briefly introduced in Sect. 3.1): queue and partial order.

The queue meta-specification represents (models) the buffered execution of subsystem actions on a subsystem driver. An action is enqueued as a result of executing the LACE program, and dequeued when this action is actually executed by the subsystem. We visualize a queue for each subsystem participating in a LACE program as a column showing all elements of the queue (see (1) in Fig. 1(b)). We position the queues above the corresponding subsystems and let them 'grow' downwards: a new element is added to the bottom of the column, and all elements are shifted up when an old element is removed. In the screen shot in Fig. 1(b), the queues of `Laser` and `Projector` are empty, the queue of `Sensor` has one element, and the queue of `Handler` has two elements.

The partial order meta-specification represents (models) a mutual dependency between subsystem actions (imposed by the control or data flow of a LACE program). Such a dependency influences the throughput of a program and in the combination with the slow execution on a subsystem can cause a bottleneck. Therefore, we visualize the partial order in the interaction with the subsystem queues (where dequeuing can model such a slow execution). For this visualization we use color highlighting ((2) in Fig. 1(b)): red rectangles highlight subsystem actions that are blocked from the execution, and green rectangles highlight subsystem actions that are being executed (*i.e.* are situated at the front/top of the corresponding queues).

To be able to execute the underlying Event-B specification of a LACE program we use buttons integrated into its visualization ((3) in Fig. 1(b)). Such buttons get enabled and disabled based on the current state of the execution, letting a DSL user "discover" the DSL semantics. Each button is attached to

[4] Note, that although the graphical notation of LACE is based on UML activity diagrams, the dynamic semantics of LACE does not follow the semantics of UML activity diagrams.

a graphical element of the LACE program in such a way that the user can intuitively associate the enabled or disabled behavior with the corresponding concept of the LACE semantics. For example, each subsystem in Fig. 1(b) has two corresponding buttons attached to its graphical representation (column): "Request SS action" (for enqueuing an action) and "Execute next SS action" (for dequeuing an action).

3.3 Generating LACE Visualizations

The resulting visualization mimics an (arbitrary) LACE program, and thus, needs to be constructed for each concrete LACE program. To make such a visualization design feasible, we automate construction of LACE visualizations using a model-to-model transformation from LACE to BMotion Studio. The transformation derives graphical elements and their layout from a DSL program and generates the corresponding observers (buttons) and controls (shapes, labels, and their attributes) of a visualization. Moreover, the LACE-to-BMS transformation connects elements of the visualization with elements of the underlying Event-B specification.

A BMotion Studio visualization runs on top of an Event-B specification. This means that controls and observers of the visualization query variables and events of the underlying Event-B specification. As the visualization design is determined (shaped) by the concrete syntax rather than by the dynamic semantics of the DSL, querying data from the Event-B specification includes rather non-trivial predicate and value expressions.

For example, to realize highlighting of subsystem actions in green and red (as depicted in Fig. 1(b)), we need to specify two observers for each subsystem action of a LACE diagram. Figure 2 shows the BMotion Studio wizard configuring such observers: an observer sets the background color to red or green when the corresponding predicate is true. The predicates check the mutual dependency between subsystem actions and the execution state of the corresponding subsystem queue. Obviously, writing such predicates manually for all subsystem actions of the LACE program is quite tedious.

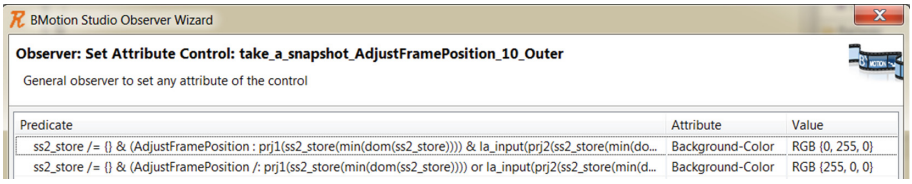


Fig. 2. Screen shot of the BMotion Studio wizard for configuring an observer

Buttons (controls) do not map one-to-one on the events either. As described above, to make semantics of buttons intuitively clear to DSL users, we attach

them to the graphical elements of a LACE diagram. For example, in Fig. 1(b) there are eight buttons attached to the four subsystem columns, and thus, representing behavior of the subsystems. However, due to another aspect of the LACE semantics, which we do not discuss here, such a behavior of a subsystem can be executed in multiple events and in the combination with the same behavior of another subsystem.[5] This means, that a subsystem button should be able to trigger different events depending on the current execution state, and moreover, some of these events should be triggered by a combination of multiple subsystem buttons. In other words, we need to realize a many-to-many relation between buttons and events.

The LACE-to-BMS transformation overcomes these technical challenges by specifying the mapping between the concrete syntax (shaping the visualization) and the dynamic semantics (driving the visualization) on the level of the LACE metamodel. Thus, all predicate and value expressions and many-to-many relations between elements of the visualization and elements of the Event-B specification are generated automatically for each concrete LACE program.

The scheme of the LACE-to-BMS transformation is depicted in Fig. 3. Within the LACE development environment, a graphical LACE program is parsed into the corresponding LACE *essential model* (*i.e.* abstract syntax tree, AST). This model is used as an input for the LACE-to-Event-B transformation described in Sect. 3.1 (the right part of Fig. 3). Besides a LACE model, the LACE-to-Event-B transformation takes as an input a set of LACE meta-specifications. As an output, this transformation generates an Event-B specification of the LACE program (bottom right corner of Fig. 3) and the corresponding mapping information. The latter is in fact the log of a transformation execution and captures the links between the elements of the resulting Event-B specification and of the original LACE model. The mapping information is necessary for connecting LACE visualizations with the underlying Event-B specifications.

A LACE visualization is generated from the original graphical LACE program (the left part of Fig. 3), as the essential model abstracts from (thus, leaves out) the notation details of the LACE diagram. The LACE-to-BMS transformation uses an intermediate visualization model, which splits the transformation into two separate steps. The first step captures graphical design of the visualization: graphical elements and their layout. The second step takes care of the mapping between Event-B events and variables and BMotion Studio observers and controls. For example, in the second step of the transformation concrete elements of the Event-B specification are substituted into predicates of the observers and controls. In this way we modularize the transformation and decouple the DSL from the visualization platform, allowing for potential reuse of our approach for employing other visualization platforms.

[5] One can observe that in Fig. 1(c) "request_ssa_ss1" appears four times in different combinations with other subsystems ("ss2" and "ss3").

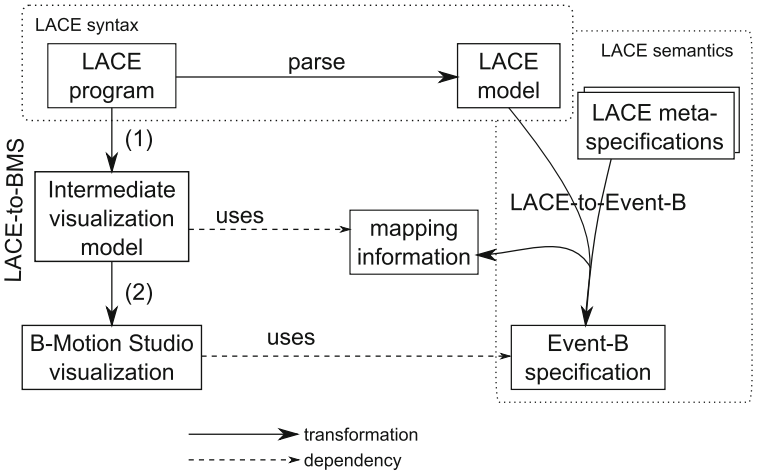


Fig. 3. Transformations from LACE to Event-B and BMotion Studio

4 Discovering Opportunities: User Study

To validate the visualization design and to discover use cases for its application, we performed a user study among LACE end-users (ASML engineers who develop software using LACE). The user study consisted of interviews (with an engineer) and of brainstorming sessions (with few engineers at once). During such an interview or a brainstorming session, we would demonstrate our LACE visualization by animating five existing LACE programs and then collect the feedback using the questionnaire form and personal communication. Ten ASML engineers participated in the user study,[6] with different expertise in LACE and from various application and/or system subdomains.

During the user study we were aiming to achieve the following goals:

- to validate our rationale for designing a DSL visualization;
- to assess how well the level of details is balanced;
- to discover opportunities for applying the visualization in the development process.

Using the GQM (Goal, Question, Metric) method [11], we refine these goals to a more operational level as a set of questions which we need to answer in order to conclude on the results of the user study. Such questions allow for interpreting the data collected during the study. In Table 1 these study questions are shown in the second column next to the goals that they refine. The questions that were asked to the LACE end-users correspond to the Metric category of the GQM method. We show (examples of) these end-user questions in the third column

[6] Note that as LACE is a programming language specialized to a specific domain, the community of LACE users is not big. We estimate that the response rate for our user study was 50%.

Table 1. The structure and the results of the user study

Goals	Questions (subgoals)	Metrics (questions asked to the end-users)	Outcome/feedback
Validate the visualization design	Design is intuitive and understandable	Do you understand the visualization of the LACE model?	Yes (3.3 out of 4)
	Observers represent the execution state	Is the visualization intuitive to find the desired information?	Yes (2.9 out of 4)
	Semantics of controls (buttons) is clear	Is the visualization intuitive to execute the desired actions?	Yes (2.9 out of 4)
	Choice of the semantics concepts	What part of LACE should be visualized?	two more concepts were proposed
Assess the level of the details	Overview of an animated diagram	Are you happy with the level of details?	Rather no than yes (1.8 out of 4)
	Insights provided by the visualization	Is highlighting of blocked and queued actions helpful?	Yes (3 out of 4)
		Do queues help to see which actions need to be executed?	Yes (3 out of 4)
	Insights missing in the visualization	Would you find a log of all processed SS actions helpful?	Rather yes than no (2.6 out of 4)
		Would you find the visualization of the data values helpful?	Indifferent (2.1 out of 4)
Discover opportunities for applying the visualization	There is a lack of support for understanding the LACE semantics	Did you have problems learning LACE?	Four people – yes
		Do you still have problems understanding some LACE models?	Mostly not, one person – yes
	The visualization can close this gap	Would this visualization help understanding those models?	Yes (2.9 out of 4)
		How much time would this kind of visualization save you?	15–20%
	What are potential use case scenarios?	How would you apply the visualization in practice?	Prototyping and validation, impact analysis, replay of execution logs and predefined sequences

of Table 1. The corresponding feedback of the users is presented in the rightmost column. Most of the questions in our questionnaire form were closed questions with a grade in the interval [0..4] to indicate the certainty and/or relevance of an answer. In Table 1 we indicate the mean value of the answers, given by the respondents to the corresponding question.

Based on the results of the user study presented in Table 1 and on other feedback of the LACE users, we draw the following conclusions.

- The visualization design is in general acceptable, but can be improved using the feedback provided by the LACE users.
- The approach would benefit from the possibility to configure the level of details of the LACE visualization. For example, a LACE user might want to specify such a configuration for the LACE-to-BMS transformation and in this way adjust the level of details of the generated visualization.
- The LACE users believe that they can use the visualization for understanding and testing behavior of their LACE programs; for replaying real-life system executions; and for checking changes after LACE gets updated (impact analysis).

Moreover, according to the LACE end-users, a crucial requirement of the proposed approach is that the DSL formal specification should be consistent with the actual implementation of the DSL. Without this consistency DSL end-users cannot benefit from the visualized animation of DSL specifications. In Sect. 5 we propose two ideas, that, among other things, allow for realizing this requirement.

5 Future Work: Applying and Reusing LACE Visualizations

Following the positive and constructive feedback of the user study, in this section we elaborate on the potential possibilities for (1) generalizing our approach and making it reusable for other DSLs and for (2) applying our visualization to implement domain-specific debugging. In Sect. 5.1 we propose an idea of reusable specification and visualization templates. In Sect. 5.2 we describe the trace framework that allows for bridging technologically diverse platforms (such as the animation of an Event-B specification and the execution on an ASML machine) through execution traces.

5.1 Specification and Visualization Templates

As described in Sect. 3.1, the dynamic semantics of LACE is defined as a set of meta-specifications and as a model transformation that instantiates and composes these meta-specifications for each concrete LACE program. Note that the examples of such meta-specifications, a queue and a partial order, are not LACE-specific and can be used for defining other DSLs. However, the LACE-to-Event-B model transformation is LACE-specific. To be able to reuse our toolset for defining the dynamic semantics of other DSLs, we need to extract the LACE-specific

information from the LACE-to-Event-B model transformation. The extracted LACE-specific information can be lifted until we find a suitable abstraction for configuring a generic DSL-to-Event-B transformation. The resulting abstraction will allow for defining the dynamic semantics of a DSL using existing meta-specifications, identified as reusable and stored in a library. In analogy with generic programming, we call such reusable meta-specifications *specification templates*. Defining the dynamic semantics of a DSL as a composition of such specification templates can facilitate construction of the DSL specification, which is coherent with the actual implementation of the DSL.

In the same way, the BMotion Studio visualizations of the meta-specifications that have been identified as specification templates, can be extracted, parameterized, and collected as their *visualization templates*. To be able to instantiate and compose these visualization templates, we need to find a suitable abstraction for configuring a generic DSL-to-BMS transformation.

An overview of the proposed generic framework is depicted in Fig. 4. This architecture generalizes the transformation scheme depicted in Fig. 3. Here the DSL visualization replaces the intermediate visualization model and specifies how visualization templates can be positioned on the top of the DSL concrete syntax in order to represent the DSL semantics definition. The feasibility of such a framework and its applicability to various DSLs require further research.

5.2 Trace Framework

In order to offer DSL developers the means to quickly implement customizable DSL tooling for inspecting *execution traces* of DSL programs, we propose a *trace framework*. The trace framework uses two kinds of execution traces: *action traces* – sequences of actions executed during the execution of a DSL program, – and *state traces* – sequences of DSL program execution states. The framework defines

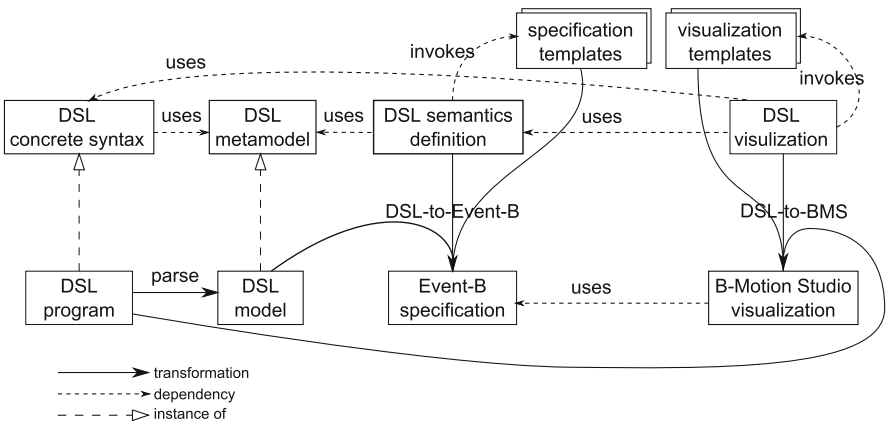


Fig. 4. A generic approach for reusing Event-B specifications and BMotion Studio visualizations

trace operations to (1) capture traces from execution logs (usually through parsing of the execution logs) and to synthesize or edit traces; (2) to visualize, replay and inspect traces; and (3) to compare traces. In order to implement DSL specific tooling, trace operations can be customized and composed into *trace applications*. To validate this approach, we implemented a number of such trace applications. Two examples are briefly described below.

The visualization of LACE programs as described in Sect. 3.2 offers users the possibility to manually play an execution scenario for a specific LACE program. As the user study showed, one of the requested applications of the visualization was the possibility to automatically replay LACE programs as they were executed on a machine. Using our trace framework, we created a trace application that reads an execution log of an actual LACE program and replays this log in ProB using the Event-B specification of this LACE program. For this, the trace application captures action traces from the LACE execution logs by parsing and reordering the data contained in the log files; and then uses these action traces to drive the ProB animator for the Event-B specification of the corresponding LACE program.

Another trace application that was created using our trace framework allows for comparing execution traces. This trace application can be used for several purposes. (1) To find a specific, for instance erroneous, behavior during the execution of a DSL program by comparing an execution trace with a trace containing the expected behavior. (2) Performing impact evaluation on two versions of the same DSL program by comparing an execution trace from a DSL program with an execution trace from a changed version of this program. (3) Finding bottlenecks in the execution of a DSL program by comparing different executions of a single DSL program. (4) To ensure that the Event-B specification of a DSL program is coherent with the generated C code by comparing execution traces obtained from ProB and from a machine.

6 Conclusion

In this work, we provide the domain-specific visualization of DSL formal specifications to support understanding and debugging of DSL programs via animation of their specifications. In general, a design of such a visualization is not obvious and is based on a fine balance between representing the underlying semantics specification in detail and being clear and intuitively understandable to DSL users. Our visualization animates the DSL diagrams by projecting on them the high-level semantics concepts, such as a subsystem buffer and an execution dependency of subsystem actions, – in contrast with the low-level semantics concepts commonly used in debugging, such as call stack, variables, and line-by-line execution. The resulting visualization does not map one-to-one on the underlying Event-B specification, which causes technical challenges in its implementation. We overcome these challenges by developing a model transformation that automatically generates a visualization (file) for each concrete DSL program.

Our user study confirms that a DSL-based development can benefit from having a formal specification of the DSL with an appropriate domain specialization of supporting tools. Following the feedback provided by the DSL users, we demonstrate that debugging of an actual DSL program (*i.e.* generated C code running on the ASML machine) is possible by replaying the execution log in ProB. As a future work, we describe our vision on how our approach can be generalized and reused for other DSLs using libraries of specification templates and the corresponding visualization templates.

Acknowledgements. We would like to thank Lukas Ladenberger (Heinrich-Heine University, Düsseldorf, Germany) for his help with using BMotion Studio. We are very grateful to all ASML engineers who participated in our user study. We also would like to thank Tom Verhoeff and Tim Willemse (both from Eindhoven University of Technology, The Netherlands) for their advice and feedback on this work and this paper.

References

1. Abrial, J.R.: Modeling in Event-B: System and Software Engineering. Cambridge University Press, New York (2010)
2. Bandener, N., Soltenborn, C., Engels, G.: Extending DMM behavior specifications for visual execution and debugging. In: Malloy, B., Staab, S., Brand, M. (eds.) SLE 2010. LNCS, vol. 6563, pp. 357–376. Springer, Heidelberg (2011). doi:10.1007/978-3-642-19440-5_24
3. Chiş, A., Gîrba, T., Nierstrasz, O.: The moldable debugger: a framework for developing domain-specific debuggers. In: Combemale, B., Pearce, D.J., Barais, O., Vinju, J.J. (eds.) SLE 2014. LNCS, vol. 8706, pp. 102–121. Springer, Heidelberg (2014). doi:10.1007/978-3-319-11245-9_6
4. Hansen, D., Ladenberger, L., Wiegard, H., Bendisposto, J., Leuschel, M.: Validation of the ABZ landing gear system using ProB. In: Boniol, F., Wiels, V., Ait Ameur, Y., Schewe, K.-D. (eds.) ABZ 2014. CCIS, vol. 433, pp. 66–79. Springer, Heidelberg (2014). doi:10.1007/978-3-319-07512-9_5
5. Ladenberger, L., Bendisposto, J., Leuschel, M.: Visualising Event-B models with B-motion studio. In: Alpuente, M., Cook, B., Joubert, C. (eds.) FMICS 2009. LNCS, vol. 5825, pp. 202–204. Springer, Heidelberg (2009). doi:10.1007/978-3-642-04570-7_17
6. Ladenberger, L., Dobrikov, I., Leuschel, M.: An approach for creating domain specific visualisations of CSP models. In: Canal, C., Idani, A. (eds.) SEFM 2014. LNCS, vol. 8938, pp. 20–35. Springer, Heidelberg (2015). doi:10.1007/978-3-319-15201-1_2
7. Mathijssen, A., Pretorius, A.J.: Verified design of an automated parking garage. In: Brim, L., Haverkort, B., Leucker, M., Pol, J. (eds.) FMICS 2006. LNCS, vol. 4346, pp. 165–180. Springer, Heidelberg (2007). doi:10.1007/978-3-540-70952-7_11
8. Mauw, S., Wiersma, W.T., Willemse, T.A.C.: Language-driven system design. Int. J. Softw. Eng. Knowl. Eng. **14**(6), 625–663 (2004)
9. Silva, R., Butler, M.: Shared event composition/decomposition in Event-B. In: Aichernig, B.K., Boer, F.S., Bonsangue, M.M. (eds.) FMCO 2010. LNCS, vol. 6957, pp. 122–141. Springer, Heidelberg (2011). doi:10.1007/978-3-642-25271-6_7

10. Snook, C., Butler, M.: UML-B: formal modeling and design aided by UML. ACM Trans. Softw. Eng. Methodol. **15**(1), 92–122 (2006)
11. Solingen, R.V., Berghout, E.: Goal/Question/Metric Method: A Practical Guide for Quality Improvement of Software Development. McGraw-Hill, Cambridge (1999)
12. Stappers, F.P.M.: Bridging formal models: an engineering perspective. Ph.d. dissertation. Chapter 6: Disseminating Verification Results, pp. 109–125. Eindhoven University of Technology (2012)
13. Tikhonova, U., Manders, M., van den Brand, M., Andova, S., Verhoeff, T.: Applying model transformation and Event-B for specifying an industrial DSL. In: MoDeVVa@MoDELS, pp. 41–50 (2013)
14. Watt, D.A., Muffy, T.: Programming Language Syntax and Semantics. Prentice Hall International Series in Computer Science. Prentice-Hall, Englewood Cliffs (1991)