Graphical simulation of the execution of DSL models

Boudewijns, R.C.

Award date:
2013

Technische Universiteit
Eindhoven
University of Technology

Department of Mathematics and Computer Science
Software Engineering and Technology Research Group

Graphical simulation of the execution of DSL models

Master's Thesis

R.C. Boudewijns

Supervisors:
dr. ir. T. Verhoeff
U. Tikhonova MSc

Public

Eindhoven, October 2013

Abstract

Using a Model Driven Engineering (MDE) approach, a Domain-Specific Language (DSL) can be used to design software on a high abstraction level. A DSL is designed using meta-modeling and its execution behavior is determined by model transformations. These model transformations bridge a big semantic gap between the high abstraction level of a DSL and an execution platform. Usually this is accomplished using complicated algorithms or design solutions which are not known by a DSL end-user. This leads to the problem that the end-users of the DSL cannot predict and understand how their models behave with respect to the execution and performance. In this report we propose to use a graphical simulation of the execution of the DSL model to overcome this problem.

A DSL that is used at ASML is used as a case study for this thesis. Its behavioral semantics are modeled using an Event-B specification which is created within the COREF project. Event-B is a specification language which employs formal mathematical notation for modeling software and/or hardware. It is supported by the Rodin platform including the ProB animator that allows the simulation of Event-B specifications. In addition, B-Motion Studio is provided as part of ProB that allows to visualize this simulation.

The transformation of the DSL to the Event-B specification language is a part of the COREF project. The research described in this thesis focuses on the transformation of the DSL to the models needed for the B-Motion Studio plug-in. This includes two research challenges: How the semantics of the DSL can be visualized, and how to achieve this visualization in B-Motion Studio using an automated approach. Using the resulting graphical simulation, a DSL user can follow execution of a DSL program and therefore reason about its behavior. It also enables a user to do an impact analysis to find out what the effects are of a change in the model. For example, an added (data) dependency between two elements in a model can have a big impact on the throughput and order of execution.

Preface

This Master's thesis is the result of my graduation project at the Software Engineering and Technology research group in cooperation with ASML. The graduation project is part of my Master of Computer Science and Engineering study at Eindhoven University of Technology.

First of all, I would like to thank Tom Verhoeff for supervising me during this project. Also, many thanks to Ulyana Tikhonova, my second supervisor, for being available whenever I had questions or needed feedback, and especially for proof reading my thesis. Next, I would like to thank Mark van den Brand and Paul de Bra for being members of my assessment committee. Furthermore, thanks go out to all ASML employees for being so readily available for discussions and feedback on the project and related matters.

And last, but certainly not least, to my family and friends for supporting me all those years.

Rimco Boudewijns,
October 2013

Contents

List of Figures

Listings

Acronyms

BMS B-Motion Studio.

COREF Common Reference Framework for Executable DSLs.

DSL Domain-Specific Language.

LA Logical Action.

LAC Logical Action Component.

LACE Logical Action Component Environment.

MDE Model Driven Engineering.

SS Subsystem.

SSA Subsystem Action.

Glossary

ASML

A provider of lithography systems for the semiconductor industry.

Controlling Software

Software which schedules and requests logical actions.

Logical Action

A behavioral model used to combine multiple subsystem actions.

Logical Action Component

A model containing a group of logical actions which can be requested by controlling software.

Subsystem

An element of a target system that provides subsystem actions. Many subsystems are a representation of a physical part of a system.

Transformation

A function mapping input information to output information. For example model transformations and code compilation.

Chapter 1

Introduction

Domain-Specific Languages (DSLs) are considered to be very effective in software development and are widely adopted by industry nowadays. A DSL meta-model defines the key concepts of a particular domain, such as major entities and their relations. A DSL improves the software development process in two ways. On one hand, a DSL captures domain knowledge and supports its reuse which raises the abstraction level of solving problems in a domain. On the other hand, a DSL implements the domain concepts and their behavior. The latter improves the reuse of design solutions and therefore raises the efficiency of the software development process.

A DSL generally consists of a meta-model and an implementation using model transformations. A DSL metamodel captures language concepts, their compositional hierarchy, taxonomy and cross references between them. Model transformations map a DSL metamodel to its execution behavior. This mapping implicitly determines the (behavioral) semantics of the model. In practice, this can cause problems while using and maintaining the DSL, because it covers a big gap between a domain level and an execution level. An explicit formal definition of the semantics facilitates the detection of errors, incoherences and efficiency bottlenecks in an early stage during the development process.

The case study for this thesis focuses on a DSL named LACE, Logical Action Component Environment, which is a mature real-life industrial DSL developed and used by ASML. ASML, a provider of lithography systems for the semiconductor industry, uses several DSLs to improve the efficiency of software development. LACE provides an environment to develop Logical Action Components which are used to generate code for the execution platform. A more detailed overview of LACE is presented in Section 2.2.1.

The (physical) subsystems referenced by a LACE model cooperate to achieve a common goal, for example, exposing a wafer. Subsystems provide operations which are combined in a LACE model with operations of other subsystems. How these operations are combined in the model can have a huge impact on the behavior of the code that is generated for the execution platform. The code generator, which is not known to a DSL user, determines how combinations of multiple subsystems are implemented. Predicting how combinations are implemented is hard for a DSL user. Therefore, ASML engineers desire to have a tool which enables LACE users to analyze LACE behavior on a high abstraction level without the need for code generation. Using a formal specification language and formal verification techniques, semantic properties can be proven, for instance, by model checking. In addition, a graphical simulation of a formal specification of the DSL can be used by a user to check if a model behaves as expected.

The COREF project [1], as described in Section 4.2.1, is meant for defining the dynamic semantics of DSLs and allows for mapping the DSL definition to the various platforms, such as verification, validation and simulation. The COREF project uses Event-B as a target language for the semantics mapping and implements this mapping using model-to-model transformation. The execution of the transformation allows for automatic generation of Event-B specifications of LACE models, and thereby for their analysis using the broad spectrum of the tools provided by

the Rodin platform.

The Event-B specifications, as detailed in Section 4.2.2, created by the COREF project employ formal mathematical notation for modeling software and/or hardware. Event-B is part of the Rodin platform which offers various supporting tools. For example, (interactive) provers, model checkers, animation frameworks and third-party plug-ins. The mathematical notation of Event-B is hard for LACE users to understand directly. To improve the usability of Event-B specifications, an interactive graphical simulation is desired. To ensure that a specification conforms to its graphical simulation, Event-B models are used directly. Because all tools use the same model, coherency is guaranteed.

Many different frameworks for the visualization of formal specifications are available. To determine which could be useful for our research, the problem domain is analyzed in Chapter 2. Next, work related to the problem domain is discussed in Chapter 3. Using the related work, the different available solutions and tools are discussed in Chapter 4. The design of the solution is explained and motivated in Chapter 5. The implementation of the chosen approach is discussed in Chapter 6. The results of this implementation are discussed in Chapter 7. Finally, the conclusion is provided in Chapter 8.

1.1 Problem Statement

The main advantage with DSLs is that they provide a high level of abstraction. However, the high abstraction level poses problems when debugging or understanding the resulting execution behavior. A user can develop models in a domain specific environment as long as the generated code implements the model as expected. However, if produced code contains abnormalities it is much harder for the user to find the root cause if the domain specific environment cannot be used to debug the model. Two main causes for abnormalities exist: The model can be wrongfully designed or the model transformation can be interpreted differently by the code generator.

The COREF project, as described in Section 4.2.1, tries to bridge the gap between the high abstraction level of a DSL and the generated code by capturing the semantics of the DSL as shown in Figure 1.1. This provides a framework to transform DSL models into a formal specification which can be used for proving, model checking and animation/simulation. The main problem of this formal specification is that LACE users find it hard to understand.

Formal specifications can be simulated graphically to improve the usability. However, LACE users do not want to create such a visualization by hand. To overcome this problem, we suggest a graphical visualization for the simulation that is created automatically.

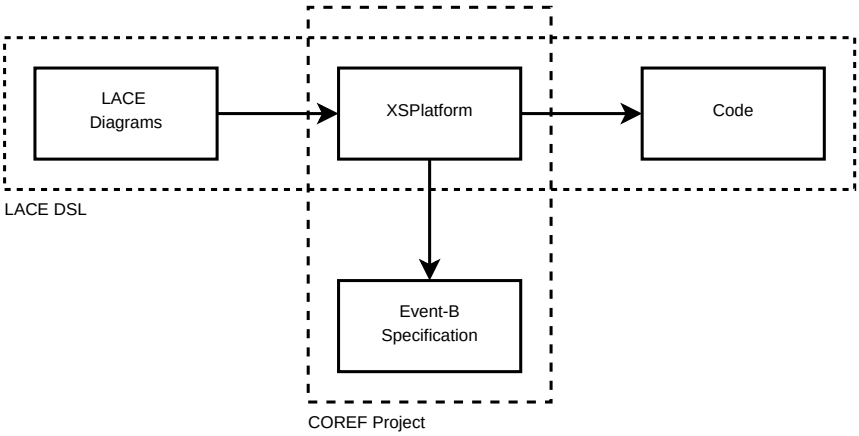


Figure 1.1: The intermediate step that is used to generate code from LACE models and how the COREF project is integrated

Furthermore, this thesis tries to investigate how transformations and intermediate models can be used to improve the reusability of model transformations. As the COREF project can be used for multiple DSLs and specification languages, the transformation of the case study should also be reusable for other DSLs and specification languages. Different specification languages use different visualization languages/tools, which implies that the transformation should also be reusable for other visualization languages/tools.

The research goals of this thesis can be captured in the following questions.

- How can transformations be optimized for reusability?
- How can intermediate steps be used and optimized for reusability?
- How can an existing DSL be enhanced using a graphical simulation?
- How can a graphical simulation improve the usability of a DSL?

Chapter 2

Problem Domain

Embedded systems are becoming more complex everyday. From a computer science perspective such systems represent a software/hardware blend at a much lower level of abstraction than usual. Adapting general purpose languages for such a domain often leads to a poor fit between the language features and the implementation requirements [2]. To improve the usability for an end-user, many domain-specific languages are developed.

A domain-specific language (DSL) is a computer language specialized to a particular application domain. Domain-specific languages are languages with very specific goals in design and implementation. A domain-specific language can be one of a visual diagramming language, programmatic abstractions, or textual languages.

2.1 System Architecture

The lithography machines produced by ASML use a Logical Action Component Environment with the layered architecture as shown in Figure 2.1. Controlling software instance determines which of the logical actions provided by the logical action components layer are requested. A logical action uses one or more subsystem operations to achieve a common goal.

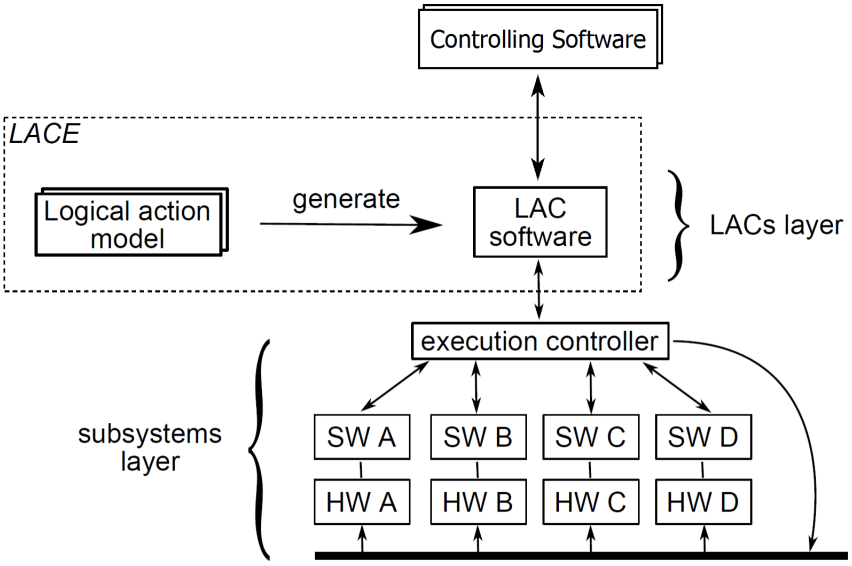


Figure 2.1: The layered architecture of the execution platform.

2.1.1 Controlling Software

Controlling software at the topmost layer determines which logical actions (LA) are requested to be executed. To ensure that subsystem actions of a logical action will not be interleaved, logical actions are requested synchronously. A controlling software instance is only allowed to request a logical action if the previous logical action is processed/queued by the corresponding logical action component. The result of the request (e.g. data) is not required to proceed with the next logical action.

2.1.2 Logical Action Components

A logical action (LA) represents a sequence of subsystem actions to be carried out by subsystems. Logical actions are created as part of a logical action component (LAC). Each LAC contains a group of logical actions that are related to each other. Logical actions are created separately but can influence each other if they share one or more subsystems.

2.1.3 Subsystems

The task of subsystems (SS) is to actually execute requests from logical actions. In general, two kinds of subsystems exist. The first kind is related to mechanical subsystems like doors. The second kind of subsystems performs measurements and calculations. Subsystems provide one or more subsystem actions (SSA). In general, subsystem actions of different subsystems can be executed asynchronously after they have been requested by a logical action.

2.2 Graphical DSLs

A graphical representation for a DSL provides its users a view which is able to provide a better overview of the involved parts. Users can create and edit a model through a graphical representation of the DSL. A serialization back-end saves the structural model separated from the graphical information. Due to this architecture, implementations that need structural and graphical information often need to process multiple files to find all information.

A graphical interface can be convenient for the user to create models, but repetitive work can be more labor intensive than by using the model directly. For example, many text editors provide a global replace operation that is often not available in graphical editors. Furthermore, graphical interfaces are less suitable for environments in which models are edited by multiple persons simultaneously. While many version control systems [3] support the merging of text based models, graphical models can prohibit the usage of these tools.

One example is the Graphical Modeling Framework (GMF) [4] which is available for the Eclipse Modeling Framework (EMF) [5]. GMF is a framework which enables users to create a graphical representation for domain-specific languages created using EMF. In GMF there are two model layers, one for describing the notation elements on the diagram and another for the semantic model that the notation elements will reference. The information used to represent and persist the visual elements in GMF is referred to as the notation model, which is separated from the underlying EMF model which is referred to as the semantic model.

2.2.1 LACE

LACE is a graphical DSL which uses UML2 as a basis. UML2 is an EMF-based implementation of the Unified Modeling Language (UMLTM) OMG metamodel [6] for the Eclipse platform. LACE is used to create logical action components, as described in Section 2.1.2, which can be used for one or more execution platforms. A LACE model describes one LAC consisting of the following parts.

- A definition of subsystem interfaces that are used by the LAC.
- A definition of the logical actions that are provided by the LAC to be used by controlling software.
- A set of logical action diagrams defining the basic structure of each LA.

The logical action diagrams are the most interesting part of the model and are used for the case study. If we have a closer look at this part we can identify a structure as illustrated in Figure 2.2. Its structure adheres to the UML2 meta-model: One or more graphical diagrams which reference a single structural model.

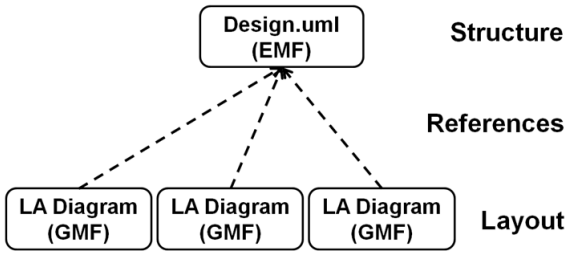


Figure 2.2: The structure of the logical action diagrams

A LAC consists of one or more logical actions which are designed in individual GMF diagrams. Such a diagram consists of vertical regions representing subsystems within a large region representing the logical action. Thick arrows define the control flow from an initial state via one or more subsystem actions to a final state. Scans are represented by yellow rounded rectangles containing one or more subsystem actions. Subsystem actions between two black bars (split and join nodes) are executed in parallel.

Furthermore, LACE supports data that can be used by subsystem actions and data transformation nodes. In addition, data can be used by the controlling software if it provided as output of a logical action. Data can be provided by the controlling software, subsystem actions and data transformation nodes. Data flow is visualized using thin arrows between these elements. Data transformation nodes are visualized using purple rounded rectangles. Examples of logical actions can be found in Chapter 7.

A LACE model created by a user cannot be used directly on the target system and should be transformed into code that can be compiled and executed. LACE models are used to create complex systems and code is generated in two steps to improve the maintainability of the DSL. The first step produces a so called XS platform model which adds additional behavioral information and discards the graphical information. This model is transformed to C++ Code and *Data Definition Files* (DDF) during a second step. Each transformation loses some graphical and/or structural information of the model.

Chapter 3

Related Work

There are many recent research efforts on embedded systems design based on MDE. Especially work involving formal specifications in model-based development can be found in the automotive industry. The reason for this is that critical embedded systems in automotive applications must operate correctly under all circumstances. Related work regarding visualizing formal specifications and model transformations can be found for a broader range of applications.

3.1 Formal Specification

Méry and Singh present a generic framework [7] for model driven development that is based on combining semi-formal notations with formal modeling language for verification purpose, correctness of the system behavior using model checker and automatic code generation from verified formal specification. Code generation from a verified Event-B formal model to code is supported by the EB2C tool [8, 9]. This tool is claimed to be easily adapted in any domain. Furthermore, it claims to give freedom for developers to adjust at best their integer representation for overcoming memory related problems and provides a mechanism to ensure the correctness of generated code. The EB2C tool is also developed as a plug-in for Rodin tool under the Eclipse framework. Though this gives some valuable information on creating a generic framework, the presented approach is not feasible for the case-study because it would require to switch to UML-B [10] instead of UML2.

Using a similar approach, de Sousa, Snook and Silva present a proposal [11] for extending UML-B to support a conceptual model, which can be better used as the starting point to pass from requirements to the abstract formal model. Once more, this approach requires to change the design of the DSL which is not acceptable. Other approaches facing the same problem are proposed [12, 13] by Esterel Technologies.

To ensure a given system really does what it is intended to do, Nascimento et al. present [14] a Model Driven Engineering (MDE) approach for the automatic generation of a network of timed automata from the functional specification of an embedded application. This is proposed because an exhaustive test of all possible system executions, or of at least a set of representative ones, is an impractical or even impossible approach for these complex systems. However, this approach is hard to use for LACE because of its specific nature which is incompatible with the UML format proposed by Nascimento et al.

3.2 Simulation and Visualization

Simulation models are means to analyze the behavior of complex processes and has been successfully applied in many industries. Müller and Pfahl detail [15] on how these models can help to find defects and how they can improve software development. For the purpose of requirement checking, the visualization technology of software requirements is one of the approaches that are used to validate the correctness of the requirements. Huang et al. present [16] a software

behavior-oriented requirements visualization method to validate the consistency between software requirements model/document and users' interpretation. By visualizing the requirements stated by a user, a simulation of a DSL model can be compared to the visualization of the requirements. However, it focuses on the requirements visualization without many details on the system visualization.

The complexity and volume of the analysis of formal specification results often prevent developers from fully taking advantage of the analysis capabilities. Goldsby et al. describe a generic visualization framework [17], Theseus, that supports a model-driven visual interpretation of analysis output from commonly used model checkers. The visualization framework supports visually interpreting the analysis results generated by model checkers in terms of the original UML diagrams. Although the framework gives some valuable insight on visualizing formal specifications, it cannot satisfy all requirements.

3.3 Model transformations

The classification that is presented by Czarnecki and Helsen [18] provides an overview of the existing model transformations. The proposed classification can be used to select a proper transformation language based on a set of properties. However, the research does not provide a concrete classification for the existing transformation languages. On the other hand, the research supports our finding that an intermediate model improves modularity and maintainability.

The COREF project [1, 19] provides a framework that can be used to create formal specifications for domain-specific languages. The framework is meant for defining the dynamic semantics of DSLs and allows for mapping the DSL definition to the various platforms, such as verification, validation and simulation. This framework is a valuable starting point for this research because it provides a methodology to create a formal specification in combination with a graphical simulation for a DSL without the need to change the DSL itself.

Chapter 4

Solution Domain

This research makes use of two main concepts of model driven engineering technologies, formal verification (in combination with visualization) and model transformations. For both concepts many implementations exist. This section provides an overview of the available implementations.

4.1 Considered Solutions

The requirements stated in Section 5.1 impose some constraints on the approaches and frameworks that can be used. The following sections detail which constraints are imposed and how these are used to select the appropriate tools.

4.1.1 Formal Verification

This work exists in the context of the COREF project and it is therefore an important factor. The current COREF implementation can only create Event-B specifications, thereby determining the specification language for the case-study. To use other specification languages one must convert the Event-B specification first. However, converting the specification is another extra step that can introduce problems and inconsistencies. Therefore, Event-B should be used as the initial specification language and a transformation to another specification language should be avoided.

4.1.2 Visualization Tools

The Event-B specification language is mainly used in combination with the Rodin framework which provides a graphical user interface that can be extended using plug-ins. A visualization framework that uses this plug-in mechanism is preferable. Furthermore the visualization language should be able to create models that resemble the LACE models as described in Section 5.1.

4.1.3 Transformation Languages

ASML uses the Eclipse Modeling Framework in combination with the Graphical Modeling Framework to create LACE models. The selected transformation language should be able to transform these models by using the meta-models/UML profiles that exist. The output of the transformation depends on the visualization framework that is selected, but Rodin plug-ins are mainly written in Java. A plug-in for Rodin provides a EMF framework for Event-B models, but not many Rodin plug-ins use it.

4.2 Formal Verification

4.2.1 COREF Project

A DSL model is often used as a source artifact which is transformed into code that can be executed on a target platform. If the DSL model is used for multiple target platforms or if the DSL is transformed into for example, a formal specification, multiple transformations are needed. The consistency between the transformations is hard to guarantee and is crucial if correctness of code should be checked using a formal specification created by a different transformation.

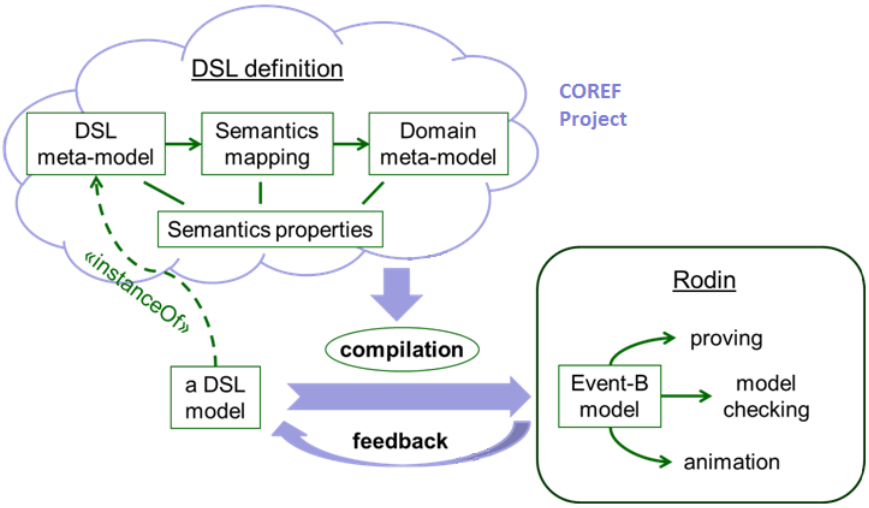


Figure 4.1: The structure of the COREF project in relation to a DSL

A way to overcome these issues is to use an intermediate step for the explicit definition of the semantics of a DSL as shown in Figure 4.1. The DSL semantics is defined via a semantic mapping from the DSL meta-model to a common semantic domain, which is then used as a source for all other transformations. As the semantics is specified once and this specification is used as a source artifact for execution, the incoherence among different DSL implementations can be detected and solved more easily. Another important advantage of this approach is an explicit definition of the dynamic semantics of a DSL, which contributes to the design development, understanding and maintenance of the DSL.

4.2.2 Event-B

Event-B is a formal method for system-level modeling and analysis using set theory as a modeling notation. A typical Event-B model consists of contexts and machines as shown in Figure 4.2. A context can be used by one or more machines and it defines an environment that a machine can use. Contexts and machines can be refined such that the initial model can be simple and abstract while the final model can be very detailed. To check the consistency between these refinements, mathematical proof obligations are generated for each model and refinement.

A context describes the static part of a model. It consists of the following.

- Constants
- Axioms
- Carrier sets

Constants are used as basic elements that are used by a context and/or machine. The type of a constant must be inferable using axioms. An axiom is a statement that is assumed to be true in the rest of the model. Each axiom consists of a label and a predicate A. All free identifiers in

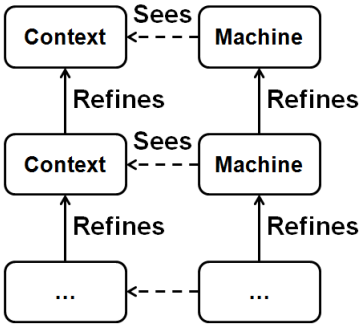


Figure 4.2: The structure of a typical Event-B model

A must be constants. If an axiom is marked as theorem it should be deductible from the other axioms. By declaring a new carrier set, it is implicitly introduced as a new constant. Commonly it is used as a declaration for an enumerated set by specifying all elements explicitly.

```
CONTEXT
    lace.testmodel.core
SETS
    SSA
CONSTANTS
    a
    b
    c
    d
    e
    f
AXIOMS
    axm1:   partition(SSA, {a}, {b}, {c}, {d}, {e}, {f}) not theorem
END
```

Figure 4.3: An example Event-B context model

An example context is presented in Figure 4.3. This example defines the `lace.testmodel.core` context using a carrier set `SSA` that contains a, b, c, d, e and f according to `axm1`.

A machine describes the dynamic behavior of a model by means of variables whose values are changed by events. It consists of the following.

- Variables
- Invariants
- Events, each consisting of:
  - Name
  - Parameters
  - Guards
  - Witnesses
  - Actions

Variables can be declared by adding their unique name (an identifier) to the Variables section. The type of the variables must be inferable by the invariants of the machine. An invariant is a statement that must be valid at each state of the machine. Invariants that are marked as theorems derive their correctness from the preservation of other invariants, so their preservation does not need to be proven.

A possible state change for a machine is defined by an event. An event can have an arbitrary number of *parameters*. The types of the parameters must be declared in the guards of the event.

The condition under which an event can be executed is also given by the guards. Witnesses are composed of a label and a predicate that establishes a link between the values of the variables and parameters of this event and the corresponding refined event. The event's action describes how the new and old state relate to each other. The initialization of a machine is given by a special event called `INITIALISATION`.

An example machine is presented in Figure 4.4. This example uses a variable `var` that contains a set of `SSA` elements according to `inv1`. During the initialization, `var` is initialized containing all `SSA` elements. Every time `event` is executed, an element from `var` is removed. The guards of `event` require that `var` is not empty.

```
MACHINE
    lace.testmodel.coremachine  ›
SEES
     lace.testmodel.core
VARIABLES
    var  private ›
INVARIANTS
    inv1:   var ⊆ SSA not theorem ›
EVENTS
    INITIALISATION:    not extended ordinary internal ›
        THEN
            act1:   var ≔ SSA ›
        END

    event:  not extended ordinary internal ›
        ANY
            e    ›
        WHERE
            grd1:   var ≠ ∅ not theorem ›
            grd2:   e ∈ var not theorem ›
        THEN
            act1:   var ≔ var \ e ›
        END

END
```

Figure 4.4: An example Event-B machine model

4.2.3 Rodin Platform

The Rodin Platform, as shown in Figure 4.5, is an Eclipse-based IDE for Event-B that provides support for refinement and mathematical proofs. The platform is open source and is further extendable with plug-ins. It provides an overview of the workspace including its contexts, machines and their contents. A context or machine can be viewed and edited using various editors either included by default or by using a plug-in.

Using multiple perspectives, different tools can use different window layouts. For example, the proving perspective provides a proof tree instead of the workspace overview. Furthermore it shows the goal of the current proof obligation in combination with controls to make the steps that are necessary to solve the proof obligation.

4.2.4 ProB Plug-in

ProB is an animator and model checker for the B-Method [20]. It is also available as a Rodin plug-in providing a new perspective as shown in Figure 4.6. This perspective enables a user to simulate the model interactively. During the simulation, the state of the model can be inspected using the variables and formulas presented. The ProB plug-in is the basis of the B-Motion Studio framework as introduced in Section 4.3.4.

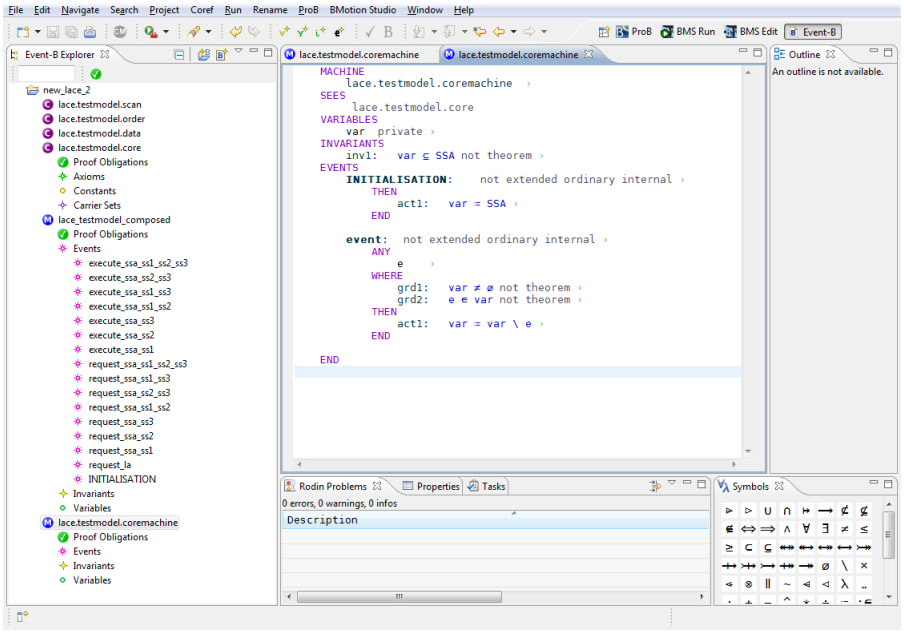


Figure 4.5: The Rodin platform showing a machine model

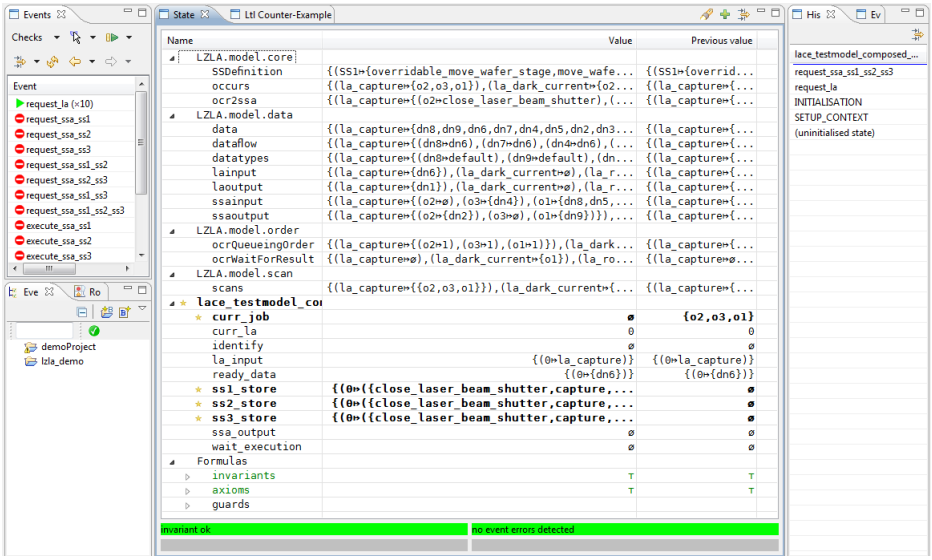


Figure 4.6: The Rodin platform showing the ProB perspective

4.3 Visualization Tools

4.3.1 B2EXPRESS

For animating Event-B models visually, [21] proposes the B2EXPRESS animator which has been developed on the basis of a data modeling technique. The animation principle consists in translating every Event-B model to a formal data model expressed in the EXPRESS formal data modeling language. However, the B2EXPRESS animator uses traces of events of the Event-B model to create an animation which prohibits the user to interact with the model easily. In addition, the B2EXPRESS tool as shown in Figure 4.7 is still in earlier stages and is not yet able to animate all specifications and no recent development has been found for this tool.

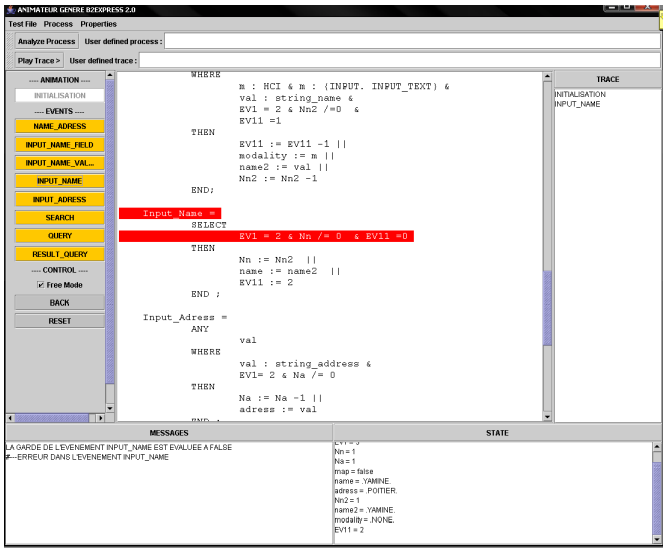


Figure 4.7: The B2EXPRESS tool

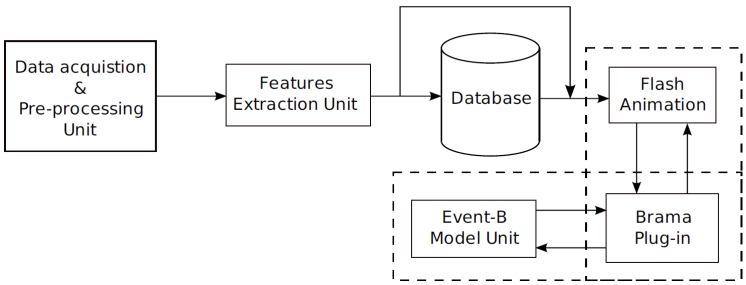


Figure 4.8: The architecture of the Brama animator

4.3.2 Brama

Brama [22] is an animator for Event-B specifications provided as an Eclipse plug-in and Macromedia Flash extension that can be used for RODIN platform. Brama can be used to create animations at different stages of development of a simulated system. To do so, a modeler may need to create an animation using the Macromedia Flash plug-in for Brama. The use of this plug-in is established through a communication between the animation and the simulation as shown in Figure 4.8. Unfortunately, the tool imposes an unwanted dependency on Macromedia Flash which is not supported (anymore) on all platforms.

4.3.3 Flash-Based Animation Engine for ProB

Another flash based tool is presented as *a generic flash-based animation engine for ProB* [23]. Using ProB as animation framework, it creates a flash animation server to serve the animation as shown in Figure 4.9. The animation server is used to interact with the model and provide state information to the graphical front-end. Each state of a B machine can be represented by a set of graphical objects such as text labels or pictures. In addition it is possible to attach a movie to a state changing operation. The tool, however, still requires the user to write some gluing code in Java to link the model and the visualization.

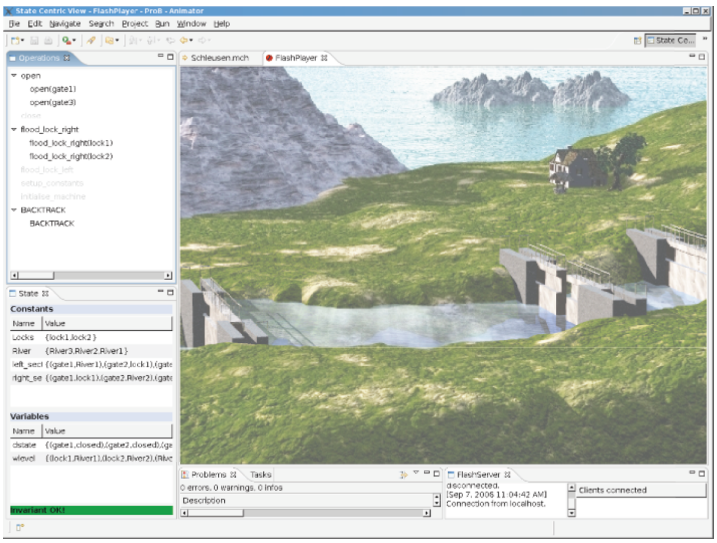


Figure 4.9: An example animation using the Flash-Based Animation Engine for ProB

4.3.4 B-Motion Studio

The same authors present B-Motion Studio [24] as shown in Figure 4.10. This framework uses a slightly different approach and comes with a graphical editor that allows a user to create a visualization within the modeling environment. While using animation functions in ProB as a basis, it is adapted very well to the Event-B modeling language. For example, a user can use Event-B notation to fill in details of a visualization.

Another advantage is that it provides a clear distinction between *Operations* and *Observers*. An operation is used to execute an Event-B event when a user interacts with an object of a visualization. An observer is used to visualize the current state of (a part of) an Event-B model by changing properties of an object. Furthermore, the framework provides extension points that can be used to create more sophisticated visualizations for domain-specific models.

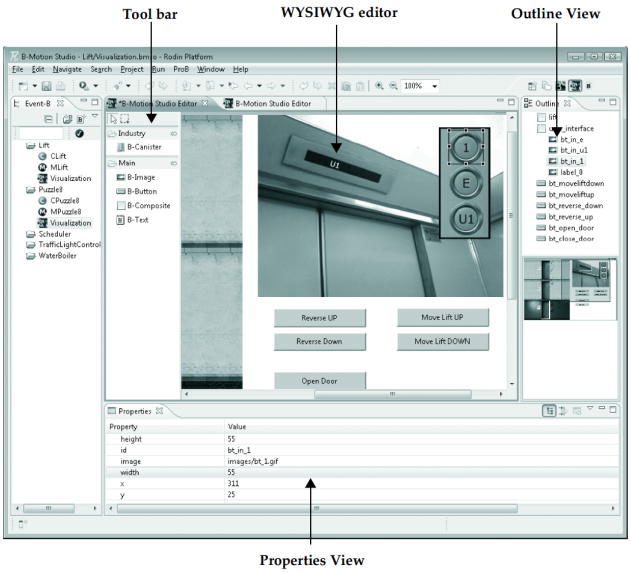


Figure 4.10: The B-Motion Studio user interface integrated in the Rodin platform

4.4 Transformation languages

Model-to-model transformation is a key technology for the Object Managemant Group (OMG)'s Model Driven Architecture. The need for standardization in this area lead to the Meta-Object Facility (MOF) Query/View/Transformation (QVT) Request for Proposals (RFP) from the OMG. This resulted in a model transformation language standard as described in [25]. It consists of three parts:

- **QVT-Operational** is an imperative language designed for writing unidirectional transformations.
- **QVT-Relations** is a declarative language designed to permit both unidirectional and bidirectional model transformations to be written. The QVT-Relations language has both a textual and a graphical concrete syntax.
- **QVT-Core** is a declarative language designed to be simple and to act as the target of translation from QVT-Relations. However, QVT-Core has never had a full implementation and in fact it is not as expressive as QVT-Relations.

The MMT project is a subproject of the top-level Eclipse Modeling Project which hosts Model-to-Model Transformation languages. Transformations are executed by transformation engines that are plugged into the Eclipse Modeling infrastructure. The following subsections present the available transformation engines that use the MMT project.

4.4.1 ATL Transformation Language

The ATL Transformation Language [26] is a model transformation language and toolkit. Although ATL does not completely adhere to the QVT standard, it provides ways to produce a set of target models from a set of source models. A model-transformation-oriented virtual machine has been defined and implemented to provide execution support for ATL while maintaining a certain level of flexibility. As a matter of fact, ATL becomes executable using a specific transformation from its meta-model to the virtual machine bytecode. Extending ATL is mainly a matter of specifying the execution semantics of the new language features in terms of simple instructions: basic actions on models (elements creations and properties assignments).

4.4.2 QVT Declarative

QVT Declarative (QVTd) is a partial implementation of the Core (QVTc) and Relations (QVTr) Languages defined by the OMG standard specification (MOF) 2.0 Query/View/Transformation. The QVTd component aims to provide a complete Eclipse based IDE for the Core and Relations languages defined by the OMG QVT Relations (QVTR) language. This goal includes all development components necessary for development of QVTc and QVTr programs and APIs to facilitate extension and reuse.

4.4.3 QVTo

QVT Operational (QVTo) [27] is a partial implementation of the Operational Mappings Language defined by the OMG standard specification (MOF) 2.0 Query/View/Transformation. In long term, it aims to provide a complete implementation of the operational part of the standard. A `mapping` maps 1..* source model elements into 1..* target element and the source and target types are indicated by mapping signature. For example, `mapping A::AtoB(): B;` maps models of type A to models of type B.

Furthermore, QVTo provides tracing of already transformed objects. For example, if the previously defined mapping is executed using `a.map AtoB();`, the newly created object can be found using `a.resolve()`. This can help to find object that are created during a transformation without the need to keep track of them.

If the declarative body of a `mapping` is unsuitable, the imperative `helper` or `query` functions may be used. A `helper` function is normally used to gather information from different types of models while a `query` is used to gather information of the same model type. In addition, a `mapping` may contain an imperative `init` and `end` part to help the declarative mapping. The `init` and `end` clauses are executed before and after the declarative part, respectively.

Chapter 5

Approach

5.1 Requirements

Before an approach can be determined, requirements should be listed. The following sections introduce the stakeholders and their desires for the case study. Furthermore, the feasibility of the case study is shown using prototypes.

5.1.1 Stakeholders

Both ASML and the TU/e are stakeholders for the case study of this thesis. On the one hand, ASML is particularly interested in the resulting visualization and how it can be used for other DSLs in their development process. On the other hand, the TU/e is interested in the design of the implementation such that it offers a generic solution for the graphical simulation of DSLs.

Marc Hamilton, one of the developers of LACE, was consulted to determine the requirements for ASML. While no strict requirements were imposed by ASML, the following requirements were determined by reviewing several prototype visualizations.

- The created implementation(s) should be reusable for other DSLs.
- An implementation should be created that generates a visualization that resembles the original model. The generated visualization should use the following design:
  - Buttons that request and execute subsystem actions should be placed above the corresponding subsystem.
  - Colors should be used to indicate that subsystems actions are requested or executed as a scan.
  - Each subsystem should have a queue with the subsystem actions that are queued.
  - It should be possible to relate the subsystem actions in the queues to the logical actions that requested them.
  - Colors should be used to indicate which subsystem actions are queued and blocked.
- A more detailed visualization of the model could also be useful but a visualization resembling the original model is easier to use by the users of the DSL.

The TU/e imposed more generic requirements:

- The created implementation(s) should be reusable for other DSLs.
- The created implementation(s) should be reusable for other specification languages.
- The created implementation(s) should be reusable for other visualization languages.
- The created implementation(s) should be configurable by an end-user.
- The created implementation(s) should be integrable into the COREF project.

5.1.2 Feasibility

To check the feasibility of the case study in an early stage, prototypes were created. The COREF project is checked by gradually transforming more LACE features into formal specifications to check if the behavior of the system can be simulated by the models. To check if the visualization of LACE is feasible, we created two prototype visualizations.

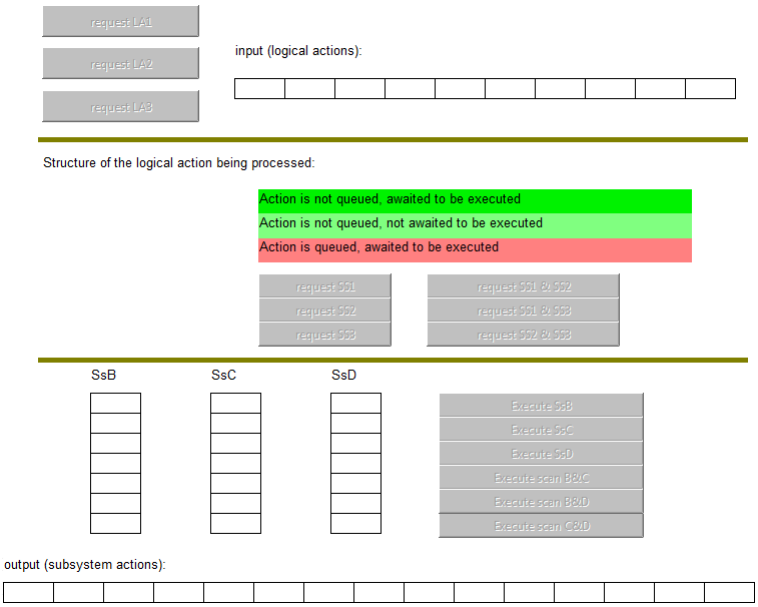


Figure 5.1: The first prototype that was already created in the COREF project to check the feasibility of a graphical simulation

The first prototype, as shown in Figure 5.1, visualizes the internals of the LACE implementation by showing a queue for each subsystem in combination with controls to perform operations. A second prototype, as shown in Figure 5.2, tries to resemble the original LACE diagrams as much as possible. These prototypes that were created using the B-Motion Studio plug-in reveal the following limitations of B-Motion Studio.

- Not all primitives used in LACE are supported, e.g. rounded rectangles.
- Not all primitives have the option to show text.
- Connections/Arrows are always drawn in a straight line, no rectilinear routing is possible.
- Connections/Arrows are forced to be connected to objects.
- Only one event can be executed upon activation of an element. Guards cannot be used to select the appropriate event.

After some research, all issues could be solved by visualizing the models a little differently. For example, rounded rectangles can be visualized by regular rectangles without losing to much information. The last issue (needed to execute scans or individual subsystem actions) was solved by using multiple buttons. More implementation details are described in Chapter 6.

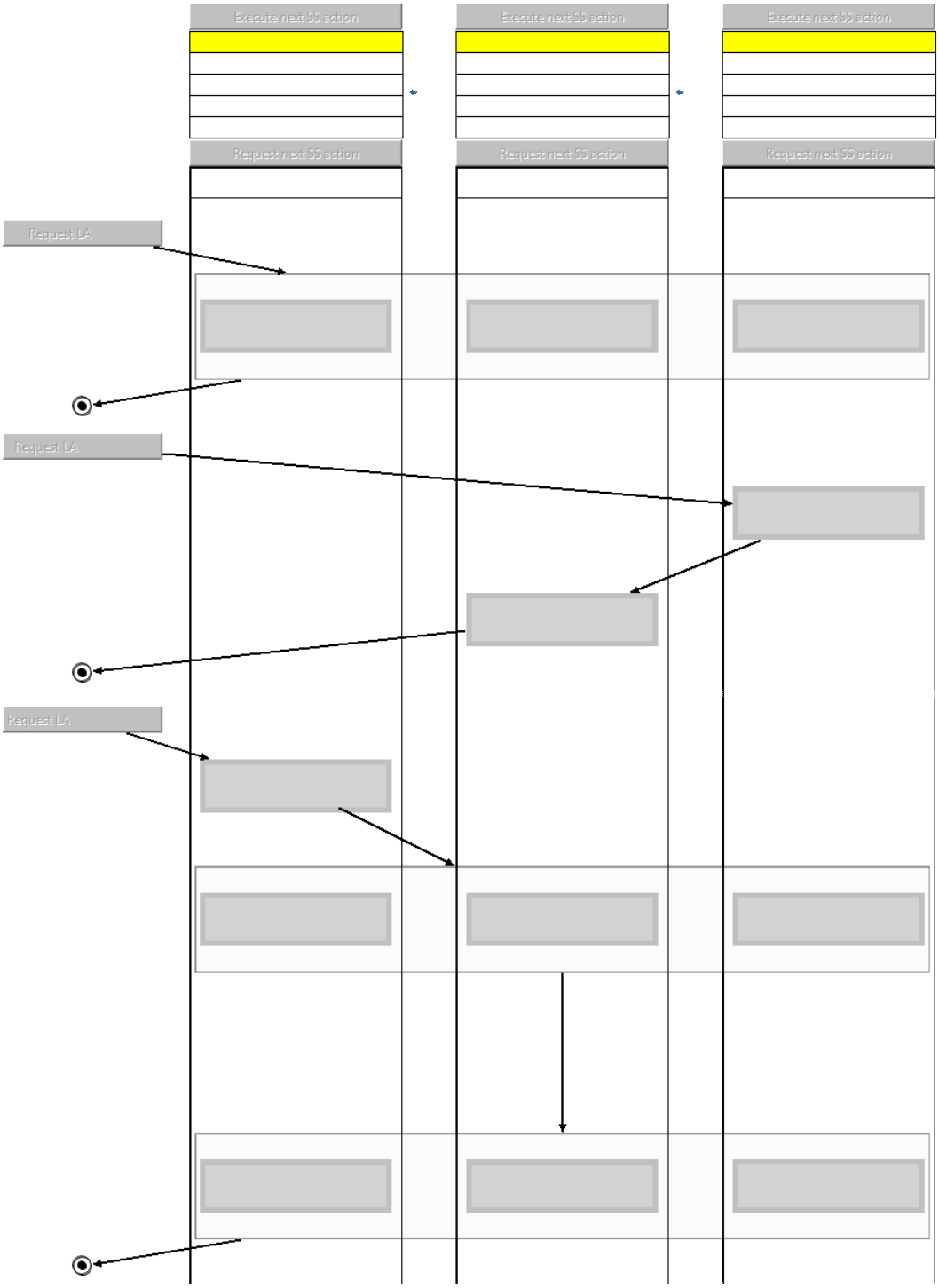


Figure 5.2: A second prototype based on an existing logical action as described in Chapter 7 that was created specifically for this project to test its feasibility and determine the requirements for a visualization

5.2 Tools

After investigating the possible solutions as described in Chapter 4, the following tools were selected:

Behavioral Information:	Event-B specification provided by COREF
Source Entity:	Original LACE model
Target Platform:	B-Motion Studio
Transformation Language:	QVTo + Java

Java was selected as secondary transformation language to create Eclipse plug-ins that can invoke the created transformations from the existing IDE. Furthermore, Java can also be used as transformation language if the source and/or target language is not compatible with EMF.

5.3 Architecture

To improve the reusability of the transformation, an intermediate model is introduced. The intermediate model is used as shown on the left hand side of Figure 5.3. It is independent from the DSL and the visualization framework if it is designed as a generic visualization model. By using two separate transformations for the source and target models, both can be changed independently without the need to change the intermediate model.

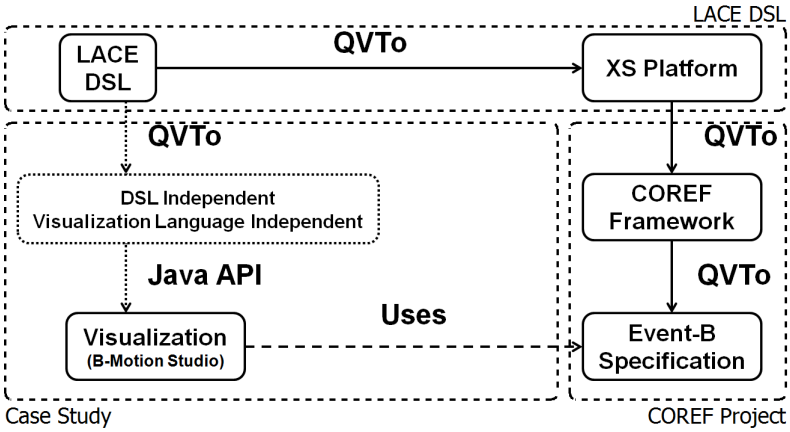


Figure 5.3: An architectural overview of our approach

5.4 LACE

While investigating LACE, we discovered that the elements with type information are stored in a *design.uml* file that is used for a whole Logical Action Component. However, the layout information is stored separately for each logical action in diagram files. To gather all information, the transformation must use the design file in combination with multiple diagram files to find all relevant elements. For each element in the *design.uml* file, the corresponding visual element must be found in one of the diagram files.

To provide a user the opportunity to select a logical action diagram as root element for the visualization, it should be used as starting point to find the corresponding node in the design. Because there exist no references from the design to the diagram files, back-references must be created by finding the corresponding nodes in the diagrams. Furthermore, the transformation should check that all selected diagrams correspond to a single design file in order to create a visualization that works with the specification created for the LAC.

The transformation that uses a LACE model as an input should be created in such a way that it supports an arbitrary number of logical action diagrams. As a starting point, the transformation can be created as stand-alone transformation, but eventually it should be integrated into the LACE development platform.

5.5 Intermediate Model

The transformation that reads the LACE models should create an intermediate model that is as generic as possible. First, it should be generic with respect to the DSL that is used as input. Second, it should be generic with respect to the visualization and specification languages that are used as target platform. By using the model driven approach provided by the EMF framework as shown in Figure 5.4, we can use model transformations to create an intermediate model from multiple DSL by relating its meta-model and the intermediate meta-model. In addition, XMI (XML Metadata Interchange) serialization can be used to save the models without the need for an explicit serialization implementation.

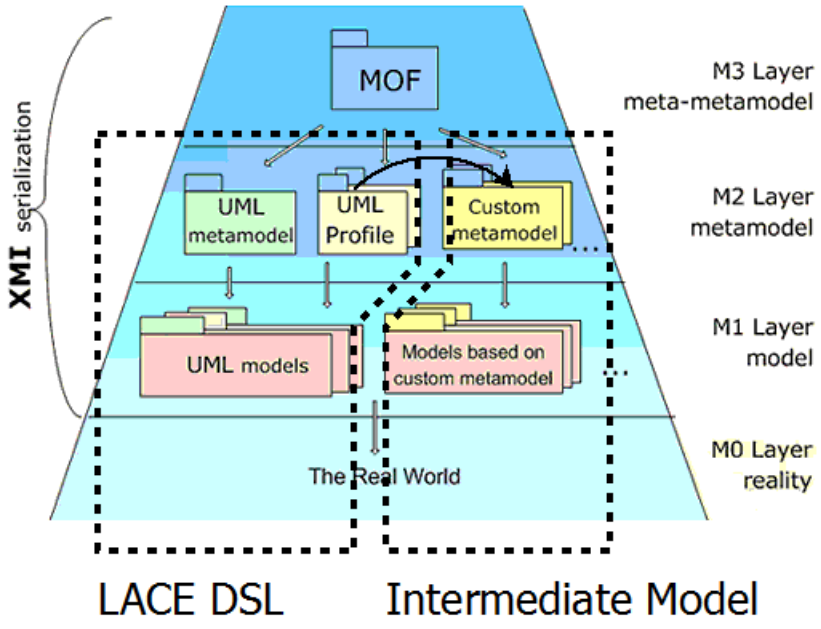


Figure 5.4: The model driven approach of the EMF framework using meta-models

5.6 B-Motion Studio

Using B-Motion Studio as target platform, we have some limitations. For example, a visualization can only contain one view that interacts with a single machine. The transformation creating B-Motion Studio visualizations should be able to create multiple visualizations or a combination of multiple intermediate visualizations to support multiple logical actions.

After researching the possible extension points of B-Motion Studio that can be used for the creation of visualization models, we discovered the following options.

- Creation of a meta-model for the visualization as none is available.
- Creation of XML files that adhere to the file format used by B-Motion studio.
- Creation of Java code that invokes the B-Motion Studio plug-in.

The first option would require a meta-model of the visualization language that is capable of creating visualization models using an EMF interface. As this would require a lot of work and will not live long, as the new version of BMS (BMS 2) is coming, this option is not viable. During the investigation of the second option, we discovered that BMS uses the XStream serialization [28]. Creating XML files that adhere to this file format would require a lot of work or the usage of the XStream serialization, both would require the implementation of a lot of functionality that is already implemented in BMS.

The developers of BMS suggested the usage of its Java interface because it is stable (BMS 2 will have a compatible interface) and can be used without reimplementing the model design or serialization from scratch. In addition, the integration with the Rodin platform is available almost for free if the Java code is created as part of an Eclipse plug-in.

5.7 Event-B

An Event-B specification created by the COREF project contains the behavioral information of a model using the formal notation. To create a visualization that can cooperate with the specification, some information should be extracted for the visualization. This information could be extracted from the formal specification by trying to map elements to the original LACE model, but a better option would be to let the COREF project provide the mapping information between the LACE model and the specification. (See Figure 5.3, right side.)

Currently, the mapping information is provided as a separate model. When this research is integrated in the COREF project, this information can be derived directly from the COREF models without the need of an intermediate step.

5.8 QVTo

As the intermediate visualization model is created using EMF, model transformations are an appropriate approach. The transformations used within ASML use QVTo to create code. By chosing QVTo as transformation language, we can use the existing transformations as example. The created QVTo implementation should use the following structure.

1. Check if the LACE model conforms to the expected structure.
2. Transform its elements using mappings as much as possible. Mappings support resolving elements that are already transformed, in contract to other function types.
3. Use helpers to acquire information that cannot be transformed using mappings.
4. The mapping information from the COREF project should be used to map LACE elements to their Event-B counterparts if needed.

The expected structure as referenced in the first point requires that the selected diagrams reference one and the same design model. The transformation that is performed after this check should also contain assertions to make sure that all elements can be found and are transformed correctly. For example, if a design model contains an element that is not present in one of the diagrams, the transformation should detect this inconsistency.

5.9 Java

The Java code that performs the transformation from the intermediate visualization model to the BMS visualization should be designed as follows.

- The user should be able to import the intermediate model using a graphical interface (GUI).
- The views of the intermediate model should be selectable by the user.
- The transformation should be able to create a visualization that uses a machine that is selected by the user.

To improve the reusability of the implementation, the code that transforms the model to the target platform should be separated from the code that provides the graphical user interface. This allows future transformations to reuse the graphical interface for multiple target platforms. For example, a transformation creating a BMS2 visualization should be able to reuse the GUI without changing a lot of code.

Chapter 6

Implementation

The approach as presented in Chapter 5 is used as a starting point for the implementation. The implementation that was created for the case-study is described in this section. First, Section 6.1 describes the architectural details of the implementation including the design of an intermediate visualization model. Next, Section 6.2 presents the implementation details of the QVTo transformation that creates intermediate models from LACE models. Finally, Section 6.3 provides more information on the implementation of the Java transformation that creates B-Motion Studio visualizations.

6.1 Architecture

Eclipse EMF is used to create an intermediate model that can be generated easily using model transformations. EMF has a distinction between the meta-model and the actual model depicted by M2 and M1 in Figure 5.4, respectively. The meta-model describes the structure of the model and a model is then the instance of this meta-model. In addition, EMF provides a framework to store the model information, by default it uses XMI to persist the model definition.

EMF allows to create a meta-model via different means, e.g. XMI, Java annotations, UML or an XML Schema. Our intermediate visualization model uses the EMF tools directly to create an EMF model. Once the EMF meta-model is specified we can generate the corresponding Java implementations classes from this model that can be used directly from Java code. Furthermore, EMF provides the possibility that the generated code can be safely extended by hand.

To be more specific, EMF is actually based on two meta-models; the Ecore and the Genmodel model. The Ecore meta-model contains the information about the defined classes. The Genmodel contains additional information for the code generation, e.g. the path and file information. The genmodel contains also the control parameter how the code should be generated.

The Ecore meta-model allows to define four types of elements:

- **EClass** represents a class, with zero or more attributes and zero or more references.
- **EAttribute** represents an attribute which has a name and a type.
- **EReference** represents one end of an association between two classes. It has flag to indicate if it represent a containment and a reference class to which it points.
- **EDataType** represents the type of an attribute, e.g. `int`, `float` or `java.util.Date`.

An Ecore model shows a root object representing the whole model. This model has children which represent the packages, whose children represent the classes, while the children of the classes represent the attributes of these classes.

The LACE-to-Event-B mapping information as described in Section 6.1.1 is used in the QVTo transformation that creates the intermediate model. Using this information, the intermediate model can be created in a DSL independent way as shown in Figure 6.1. Although an instance of the intermediate model contains specification language specific information, the structure used to store this information is specification language independent.

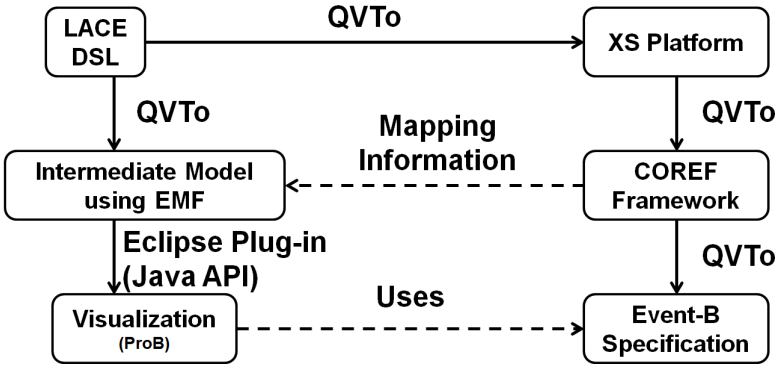


Figure 6.1: The architecture of the implementation including transformation information

Using EMF, the intermediate model as presented in Figure 6.2 is created. The root object of the model is a `visualization` which contains one or more `views`. Using multiple views, a visualization can create multiple perspectives for a single specification simulation. Each view has a `size` attribute that defines the size of the canvas that contains all `objects` of the view. Each object is of a certain `type` and is identified by a unique `id`. The available object types are specified by the `ObjectType` enumeration.

The `attributes` of an object use a special approach to create an `EMap`. An `EMap` is a data type composed of a collection of $(key, value)$ pairs, such that each possible key appears at most once in the collection. To create an `EMap` using EMF, the following steps are needed.

- Create an EClass with the name [Type1]To[Type2]Map where [Type1] represents the key's type and the [Type2] represents the value's type.
- Set the Instance Class Name property of the EClass to `java.util.Map$Entry`.
- Create an EAttribute or EReference named `key` and set the EDataType or EClass for it.
- Create an EAttribute or EReference called `value` and set the EDataType or EClass for it.

If this special class is referenced in the model, the EMF code generator will detect this special case and generate a properly typed `EMap` getter/setter instead of a normal `EList` getter/setter. To keep the attributes as generic as possible, the keys are of type `EString` and the values are of a custom `Attribute` class. The attribute class uses a basic `type` as specified by the `AttributeType` enumeration. The contents of an attribute can be stored in as `value` as long as it is a subclass of `EJavaObject`.

To represent colors in an appropriate manner, a generic `ColorRGB` class is introduced providing separate values for the red, green and blue channel. As this is a very useful data type for visualizations, it is provided in the intermediate model.

Objects can perform (multiple) `actions` if the user interacts with the object. An action contains the following attributes.

- **event** references the event that should be executed.
- **predicate** is the predicate that should hold to execute the event.
- **parameters** is used to supply parameters to the event that is executed.

Multiple actions can be created for an object, but only one action is executed upon user interaction. If multiple actions have a predicate that holds, one of them is (randomly) selected for

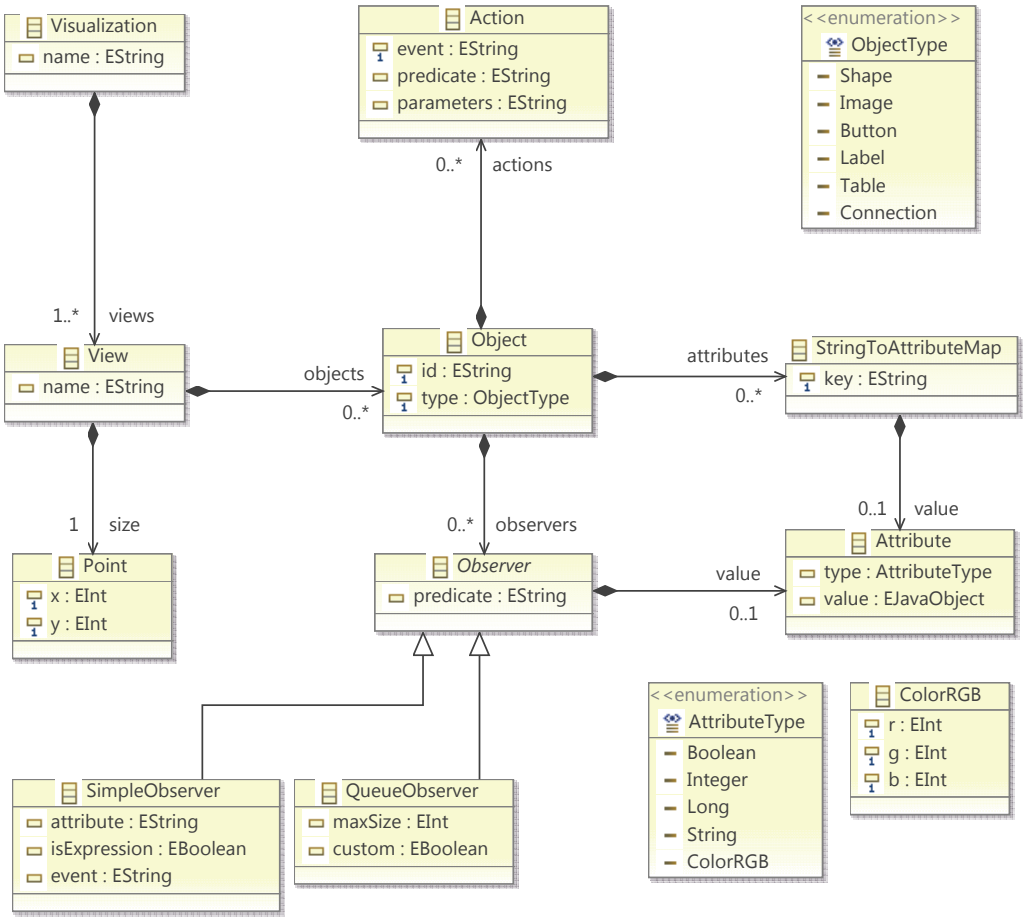


Figure 6.2: The meta-model of the intermediate visualization model

execution. If the action that is selected for execution has a guard that prohibits execution, the visualization executes no action at all.

Although the `action` class is closely related to the operations provided by the Event-B and B-Motion studio combination, it is still generic enough to be reused for other target languages which have a similar structure as long as the specification language provides events that can be invoked from the visualization.

Object represent the current state of a simulation by using `Observers`. Observers change attributes/contents of an object if the `predicate` of the observer holds in the current state. Using a combination of multiple observers, more extensive visualizations can be created. The abstract `Observer` base class can be used as extension point to create special type of observers. The current implementation provides two types, a `SimpleObserver` and a `QueueObserver`.

The behavior of a simple observer can be deducted using the flowchart presented in Figure 6.3. This type of observer uses a very generic implementation which uses one main predicate which must hold. If the `predicate` holds, `isExpression` determines how the `value` attribute is used. If

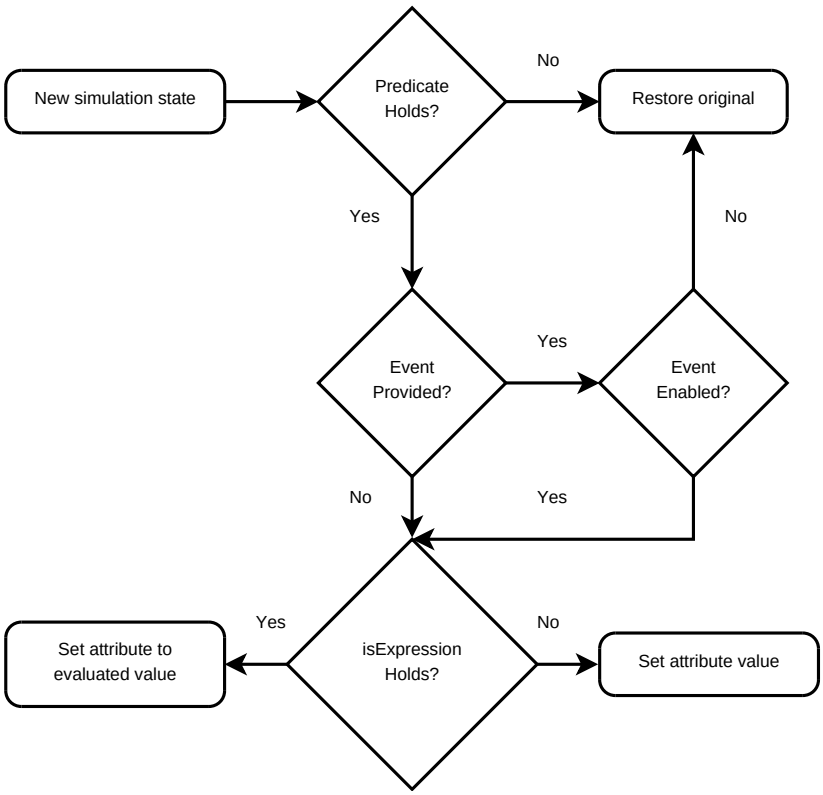


Figure 6.3: A flowchart representing the behavior of a simple observer.

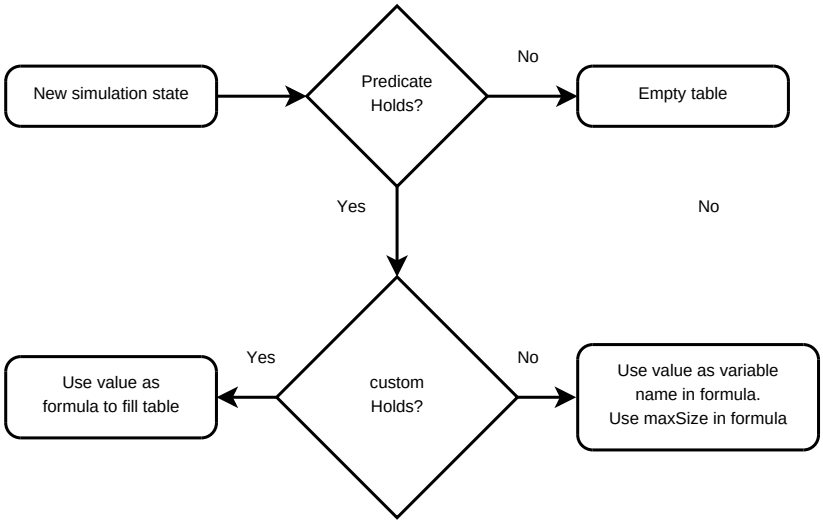


Figure 6.4: A flowchart representing the behavior of a queue observer.

`isExpression` holds, an attribute can be changed using the static `value` that is specified in the model. If `isExpression` does not hold, an attribute can be changed by evaluating the expression specified in the `value` attribute at runtime.

A queue observer provides a visualization for a queue object of type `ObjectType::Table`. A queue is an abstract data type or collection in which the entities in the collection are kept in order. Queues are used in the LACE case-study to indicate which subsystems are requested and waiting to be executed. Queues that are typed as $X \mapsto Y$ for $X \in \mathbb{N}$ and arbitrary type Y can

be visualized using the variable name as `value`. The maximum number of elements that should be visualized can be specified using the `maxSize` attribute, limiting the number of cells visualized. If a different type of queue is used, the `custom` attribute should be set to *true* and the `value` attribute should contain a formula that evaluates to a queue of type $X \mapsto Y$ using the methods provided by the visualization language. Each element in the queue is visualized as an cell in a single column. The keys of the queue are used to sort the queue in ascending order. Only the values of the queue The flowchart presented in Figure 6.4 can be used to deduct the behavior of a queue observer.

6.1.1 Mapping Information

The mapping information is captured in a model which is an instance of the meta-model as depicted in Figure 6.5. The `LogicalActionComponent` class is used as root element of the model. It contains the following information:

name	The name of the LAC in LACE for which the information holds
laRequestEvent	The event that should be used to request a LA
laRequestParameter	The parameter that should be used to set the desired LA in the request
laInputVariable	The variable that contains the queue of requested logical actions
currentLaVariable	The variable that contains the identifier of the LA that is currently active
ssDefinitionVariable	The variable that contains the set of subsystem actions provided by each SS

Furthermore, the `LogicalActionComponent` class contains two references. The `subsystemCombinations` variable references instances that provide the request and execute for a given set of subsystems. As EMF had problems creating a mapping using a set of objects as keys, the instances themselves contain the set of subsystems. To find the request or execute event for a given set, all `subsystemCombination` instances should be inspected to find the matching instance.

The second reference is provided by the `logicalActions` variable. Each instance represents a single LA containing its LACE `name` in combination with the name (`laRequestName`) that should be used as value for the `laRequestParameter`. In addition, the `subsystems` variable provides information on each subsystem used in the logical action.

Each `Subsystem` contains its LACE `name` and the name of the `queue` variable that contains the requested subsystem actions. The ordered list of `occurrences` is used to identify each individual subsystem action provided by the subsystem in Event-B. For example, if a subsystem is used in a LA to execute three subsystem actions, the `occurrences` variable should contain three strings referring to the subsystem actions in the order that they are requested in the logical action.

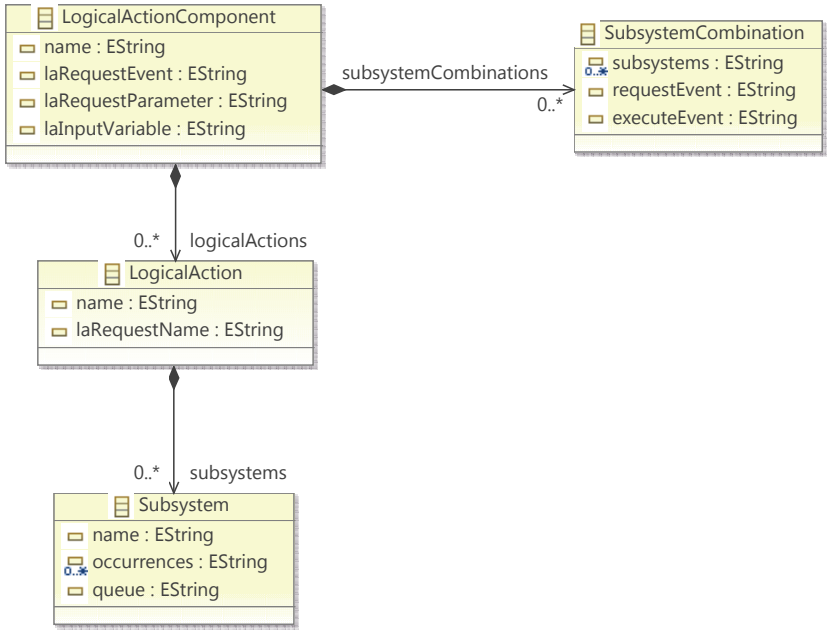


Figure 6.5: The meta-model used for the COREF transformation information

6.2 QVTo Transformation

The QVTo transformation is implemented according to the structure that is proposed in Section 5.8. An overview of the transformation is visualized in Figure 6.6 and can be read as follows.

- An input selection model is mapped to a visualization.
- The selected design.uml file containing the structural information of the LAC is used for the whole visualization.
- The selected LA diagrams are mapped to individual views.
- The elements in each diagram (stored in the design.uml file) are visualized in the corresponding view.
- The parameters of an element are used to construct the appropriate attributes of the corresponding visualization object.

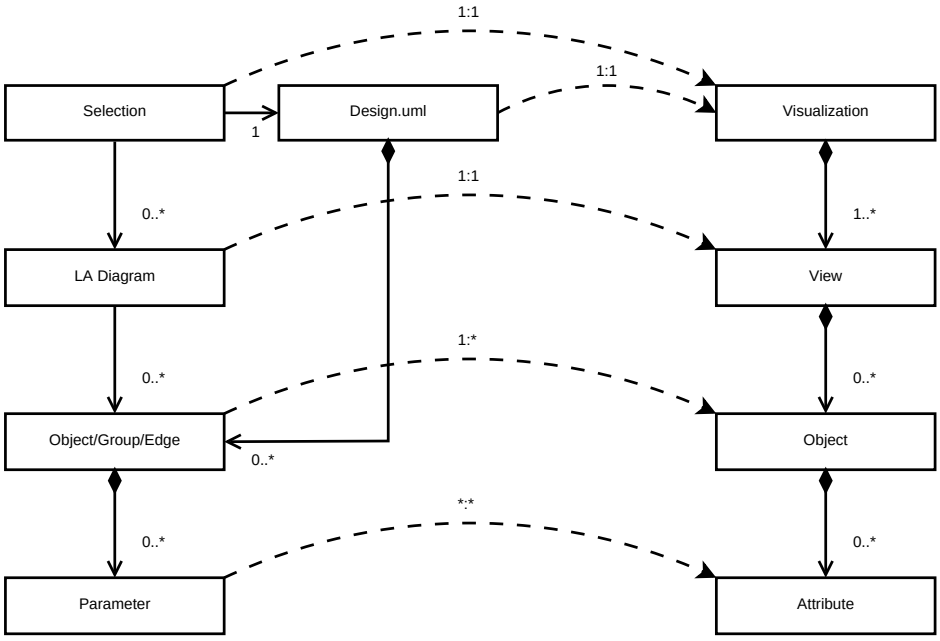


Figure 6.6: The QVTo mapping from a LACE model to an intermediate visualization model

6.2.1 Initialization

To use the meta-models that are described in Section 5.3 in the QVTo transformation, they need to be initialized as shown in Listing 6.1. In addition, this initialization provides an overview of the meta-models that are needed for the transformation. As this initialization only provides references to the meta-models, the locations of the meta-models are not important for the transformation. However, the user must make sure that the meta-models are available at runtime. Either as part of a plug-in that is loaded or by giving the absolute location of the meta-model. The mapping information model is defined as `mapping`, but because `mapping` is a keyword already used in QVTo, it must be escaped using an underscore. For example,
`modeltype _mapping uses _mapping('http://asml.visualization.lace.mapping')`. The same holds for the `intermediate` keyword.

```
1  -- Input model referencing all desired inputs
2  modeltype inputs uses inputs('http://asml.visualization.lace.inputs');
3
4  -- UML + LACE models
5  modeltype uml uses uml('http://www.eclipse.org/uml2/2.1.0/UML');
6  modeltype lace_genmodel uses lace_genmodel('http://www.lace.asml/2008/lace_genmodel');
7
8  -- GMF notation model
9  modeltype notation uses notation('http://www.eclipse.org/gmf/runtime/1.0.1/notation');
10
11 -- Intermediate model (output)
12 modeltype intermediatemodel uses _intermediate('http://asml.visualization.intermediate');
13
14 -- COREF Information model
15 modeltype _mapping uses _mapping('http://asml.visualization.lace.mapping');
```

Listing 6.1: The model definitions used for the QVTo transformation

6.2.2 Transformation Declaration

To create a transformation in QVTo, a transformation declaration must be given. A transformation declaration defines types of the input and output models that are used. Using a definition as shown in Listing 6.2, the Eclipse IDE is able to provide a user an interface in which input and output models of the correct type can be selected. Because this structure only allows a fixed number of input and output models, an input model is used to be able to select an arbitrary number of LA diagrams. The output model is an intermediate model as described in Section 6.1

The input model consists of a single class containing two references. The first reference is used to select the logical action component (design.uml) and the second reference is a list. This list can be used to select an arbitrary number of logical action diagrams. This provides the user more freedom, but as this is not an intuitive procedure, an additional GUI that provides this functionality is desirable.

Using the `_mapping` model as secondary input, information about the COREF transformation can be queried as described in Section 6.1.1

```
1  -- Transformation declaration
2  transformation transform(in Source: inputs, in Map: _mapping, out Target: intermediatemodel)
       ;
3
4  main() {
5
6      -- Check source model
7      Source.check();
8
9      -- Transform source model
10     var lacMapping := Map.rootObjects()[_mapping::LogicalActionComponent]
11                       ->asOrderedSet()->first();
12     Source.rootObjects()[Selection]->map SelectionToVisualization(lacMapping);
13 }
```

Listing 6.2: The transformation declaration and main entry point

The **main** function serves as an entry point for the declared transformation. The **main** function uses an imperative approach and should contain (calls to) everything needed to complete the transformation. Before the actual transformation is invoked, we first check if all models contain all essential information by performing input validation.

6.2.3 Input Validation

The main input validation, as shown in Listing 6.3, is used to check several essential properties. The following should at least hold.

- Exactly one root object of type `Selection` should be provided in the input model.
- This root object should reference at least one logical action diagram.
- All referenced logical action diagrams should be part of the selected logical action component.
- All referenced logical action diagrams should contain exactly one logical action. This is also a constraint for code generation, but the GUI doesn't prohibit a user from creating more than one logical actions.
- All logical actions should be available in the COREF mapping information.

The described assertions are used mainly to check if the user has selected valid models before attempting to transform them. If one of these assertions is not satisfied, the transformation is aborted. If the user has selected the models that are valid according to the input validation, these shouldn't give any problems during the actual transformation. If the transformation detects that there are still some problems, these are the result of an incorrect implementation of the transformation. For example, newer LACE versions can contain new types of elements that are not supported by the transformation. The transformation will detect that it encountered an unknown object and it will inform the user.

```
1  helper inputs::check() {
2      -- Check if the root object is valid
3      var mainObjects := self.rootObjects()[Selection];
4      assert fatal (mainObjects->size() = 1) with log('Multiple or no Selection found!');
5      var mainObject := mainObjects->asOrderedSet()->first();
6
7      -- Check if at least one logical action is selected
8      var diagramList := mainObject.diagram[notation::Diagram];
9      assert fatal (diagramList->size() > 0) with log('No logical action diagrams found!');
10
11     -- Check if the selected logical actions are defined in the selected LAC
12     var lacComponent := mainObject.lac;
13     diagramList->asOrderedSet()->checkLA(lacComponent);
14
15     -- Check COREF Information model
16     Map.check(diagramList);
17 }
```

Listing 6.3: The main validation function

6.2.4 Diagram Transformation

The entry point of the transformation that transforms a given selection is presented in Listing 6.4. The `SelectionToVisualization` mapping creates a `Visualization` that uses the name of the LAC and contains a view for each selected diagram. The `asOrderedSet()` function is used because the intermediate model requires the views to be ordered.

Views are created by transforming all selected diagrams. To optimize the data that can be used during the transformation, the `init` clause gathers some information up front:

1. The logical action that corresponds to the diagram.
2. The elements that are parts of this logical action.
3. A superset of all subsystems such that they can be used for scans.
4. The diagram element that contains all visual elements of the logical action.

The body of the `mapping` performs the following steps.

1. Set the name of the view to the name of the logical action.
2. Transform all elements (nodes, groups and edges) to objects as described in Section 6.2.5 and combine them as one large set of objects.
3. Calculate the size of the used canvas (containing all objects).
4. Create a border to visualize the size of the canvas.
5. Recalculate the size of the canvas including the border.

All objects are positioned 200 pixels from the top to keep room for the queues and buttons that are placed above each subsystem. The border of the logical action is also adjusted to take this into account.

The `end` clause is used to update the height of the subsystem areas. As the visual element representing a subsystem is of infinite height, no height is available in the model. To simulate infinite height, the element is stretched to the border of the logical action. However, this can only be done after all elements are created and the border is available.

```
1
2  mapping Selection::SelectionToVisualization(in lacMapping: _mapping::LogicalActionComponent)
3      : Visualization {
4
5      name := self.lac.name;
6      views := (self.diagram->map DiagramToView(lacMapping))->asOrderedSet();
7  }
8
9  mapping Diagram::DiagramToView(in lacMapping: _mapping::LogicalActionComponent)
10     : intermediatemodel::View {
11
12     init {
13         -- Get the LA in the design.uml file and extract all elements
14         var la := self.LA();
15         var laNodes := la.node->asOrderedSet();
16         var laEdges := la.edge->asOrderedSet();
17         var laGroups := la.group->asOrderedSet();
18
19         -- Create a powerset of the subsystems for the scans
20         var ssPowerset := createSSPowerset(laGroups[ActivityPartition]);
21
22         -- Find the element in the diagram that corresponds to the logical action
23         var laRef := self.findLAReference(la);
24     }
25
26     -- Set all properties
27     name := la.name;
28
29     -- Create all objects using the three types of elements found
30     objects :=  laNodes->map ActivityNodeToObjects(lacMapping, self, laNodes,
31                 object Point{x:=0; y:=200})->flatten()->asOrderedSet()->
32
33                 union(laGroups->map GroupToObjects(lacMapping, ssPowerset, self, laGroups,
34                 object Point{x:=0; y:=200})->flatten()->asOrderedSet())->
35
36                 union(laEdges->map EdgeToObjects(lacMapping, self, laEdges,
37                 object Point{x:=0; y:=200})->flatten()->asOrderedSet())->asOrderedSet();
38
39     -- Calculate the size that contains all objects
40     size := maxCoordinates(result.objects);
41
42     -- Create a border that represents the logical action
43     objects += object Object{
44                 id := result.name + '_Container';
45                 type := ObjectType::Shape;
46                 attributes := la.LA_Container_Attributes(laRef,
47                     object Point{x:=0; y:=200},
48                     object Point{x:=result.size.x + 20; y:=result.size.y-180});
49             };
50
51     -- Recalculate the size including this new object
52     size := maxCoordinates(result.objects);
53
54     end {
55         -- Update the heights of the subsystems (initially unknown/infinite)
56         -- such that they stop at the border
57         updateSSHeights(laGroups[ActivityPartition], result.size.y);
58     }
59 }
```

Listing 6.4: Diagram transformation implementation

6.2.5 Element Transformation

Each type of element is transformed to the corresponding objects in an intermediate model. Most elements are visualized using a simple object, but some need multiple elements. For example, a subsystem consists of multiple rectangles. One rectangle that is used for the subsystem area and a second rectangle on top for the caption of the subsystem.

Many elements use the same basic principle that is shown in Listing 6.5 consisting of several steps:

1. Add a new object to the result set, including:
   - the global identifier of the object (including a unique number to prevent duplicates),
   - the type of the object,
   - the attributes of the object as detailed in Section 6.2.6,
   - the actions and observers of the object (not shown in the example).
2. Indicate that the element is processed such that we can check if all elements are recognized.

```
1  -- Activity parameter nodes (multiple):
2  if (self.oclIsTypeOf(ActivityParameterNode)) then {
3      result += object Object{
4          id := laName + '_ActivityParameterNode_' + number.toString() + '_Shape';
5          type := ObjectType::Shape;
6          attributes := self.ActivityParameterNode_Shape_Attributes(ref, relativeTo);
7      };
8      processed := true;
9  } endif;
```

Listing 6.5: An example element (activity parameter node) transformation

6.2.6 Element Attributes

Attributes of objects are dynamic and no fixed set of attributes is defined in the meta-model. However, to improve consistency, the following set of attributes is standardized for all objects:

Attribute	Description	Default value
x	The x coordinate of the object	*compulsory*[1]
y	The y coordinate of the object	*compulsory*[1]
width	The width of the object	0[1]
height	The height of the object	0[1]
visible	Whether the object should be visible at runtime	`true`
enabled	Whether the user should be able to interact with the object	`true`
z-order	The Z-order of the object. An object with a higher Z-order will be shown on top of objects with a lower Z-order	0
text	A textual label shown on the object[2]	*empty*

Omitting an attribute implies that the default value should be used. The specified value (given or by default) is used at the start of a simulation. However, observers can change attributes during the simulation.

[1] `Connection` objects may omit this value if they contain a source and target attribute
[2] Not supported by `Image` objects

Furthermore, some types of objects use additional attributes that are standardized. `Shape` objects use the following additional attributes:

Attribute	Description	Default value
shape	The shape of the object	*compulsory*
background-alpha	The visibility of the background (0-255)	`255`
background-color	The color of the background	`white`
outline-alpha	The visibility of the outline (0-255)	`0`
outline-color	The color of the outline	`black`

`Connection` objects provide source and target attributes that can be used instead of absolute placement using coordinates. The complete list of additional attributes is as follows:

Attribute	Description	Default value
line-width	The line width of the connection	`1`
source	The id of the object that is used as source of the connection	*none*
target	The id of the object that is used as target of the connection	*none*
arrow-source	A boolean indicating whether the line has a arrow at the source side	`false`
arrow-target	A boolean indicating whether the line has a arrow at the target side	`false`

`Table` objects use additional attributes which also provide means to set attributes of individual columns and cells. Attributes of columns and cells can be accessed using zero indexed values. The complete list of additional attributes is as follows:

Attribute	Description	Default value
rows	The number of rows in the table	`0`
columns	The number of columns in the table	`0`
column_X**_width**	The width of the column X	`0`
cell_X**_**Y**_text**	The text of the cell at column X and row Y	*empty*
cell_X**_**Y**_text-color**	The text color of the cell at column X and row Y	`black`
cell_X**_**Y**_background-color**	The background color of the cell at column X and row Y	`white`

As many objects that are created during a transformation use a quite large list of attributes, they are separated from the transformation process. All attributes that need to be set for an object are combined in one function. An example of such a function is given in Listing 6.6. This example uses a `ActivityNode` and returns a list of attributes. The `ref` parameter provides access to the visual element that corresponds to the provided structural element. The `relativeTo` parameter is used to shift the object relatively to the canvas. This is mainly used to shift all object in vertical direction to create space for extra objects at the top of the visualization.

Elements of the diagram can have a width or height of -1 indicating that none has been specified by the user. This special case is identified during the transformation resulting in a default width and/or height. The `addAttr` function is a helper function to create and add an attribute using a short notation.

```
helper ActivityNode::ActivityParameterNode_Shape_Attributes(in ref: notation::Node,
    in relativeTo: Point) : OrderedSet(StringToAttributeMap) {

    var res : OrderedSet(StringToAttributeMap) := OrderedSet{};

    var width := ref.layoutConstraint.oclAsType(notation::Bounds)->width;
    if (width < 1) then { width := 100 } endif;

    var height := ref.layoutConstraint.oclAsType(notation::Bounds)->height;
    if (height < 1) then { height := 20 } endif;

    res := addAttr(res, 'shape', 'Rectangle');
    res := addAttr(res, 'x', relativeTo.x+ref.layoutConstraint.oclAsType(notation::Bounds)->
        x);
    res := addAttr(res, 'y', relativeTo.y+ref.layoutConstraint.oclAsType(notation::Bounds)->
        y);
    res := addAttr(res, 'width', width);
    res := addAttr(res, 'height', height);
    res := addAttr(res, 'background_color', object ColorRGB{r:=225; g:=225; b:=135});
    res := addAttr(res, 'outline_color', object ColorRGB{r:=0; g:=0; b:=0});
    res := addAttr(res, 'outline_alpha', 255);
    res := addAttr(res, 'text', self.oclAsType(ActivityParameterNode).name);
    res := addAttr(res, 'z_order', 5);
    return res;
}
```

Listing 6.6: An example element (activity parameter node) attribute transformation function

6.2.7 Actions and Observers

If an element needs to have an action, the transformation creates an action instance. This instance is than added to the list of actions of the corresponding object. Observers use a similar approach. An observer instance is created and added to the list of actions of the corresponding object.

For example, Listing 6.7 shows the creation of an action instance which executes the event that is specified in the `foundName` variable upon activation. Next, the example shows the creation of an observer for the subsystem request button. This observer uses the same event specified in the `foundName` variable to change the `enabled` attribute to true if the event is enabled.

Subsystem buttons are created for each subsystem used in each logical action. To check if the subsystem button should be enabled, the predicate is used to verify if the correct logical action is currently active.

```
    buttonObject.actions += object _intermediate::Action{event:=foundName};

    buttonObject.observers += object _intermediate::SimpleObserver{
        event:=foundName;
        attribute:='enabled';
        value:=makeAttr(true);
        predicate:=lacMapping.laInputVariable + '(' + lacMapping.currentLaVariable + ') = '
            + laRequestName
    };
```

Listing 6.7: The creation of an action and an observer for a subsystem request button

6.3 Java Transformation

The Java transformation consists of several classes of which two are important. The class that provided the user interface as part of an import wizard and the class that performs the actual transformation. The following sections detail these two classes.

6.3.1 User Interface

The created plug-in provides a user interface as shown in Figure 6.7. The interface provides the following functionality.

1. The project folder in which the visualization is created can be selected. If the import functionality was invoked from a project folder, it will be used as default.
2. An Event-B machine in the project folder can be selected. This machine will be used to simulate the visualization.
3. The intermediate file that should be transformed can be selected.
4. One or more views of the intermediate file can be selected.
5. The views can be combined in one B-Motion Studio visualization. Otherwise multiple visualization files will be created.
6. The name of the resulting visualization file can be set. If multiple visualizations are created, it will be used as prefix.

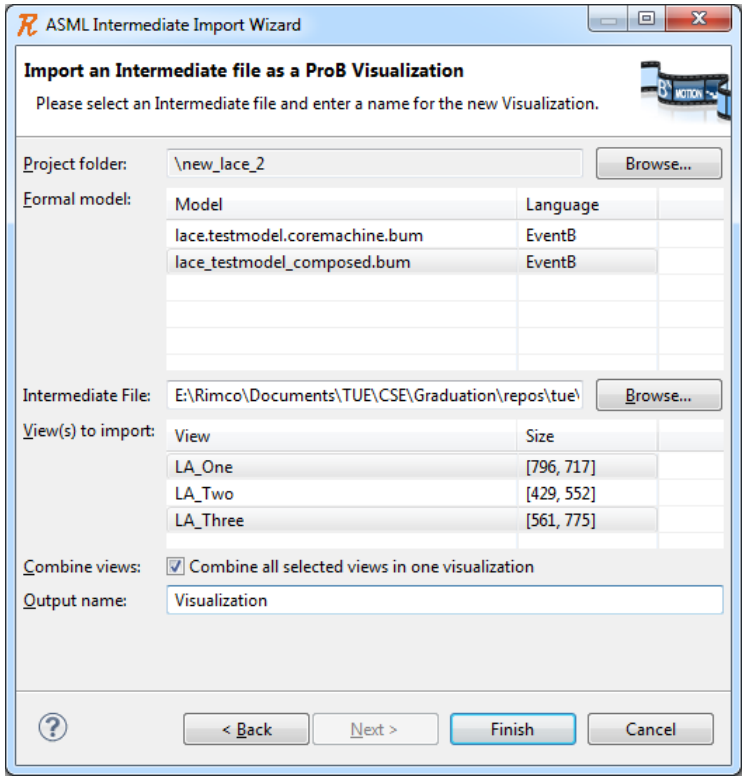


Figure 6.7: The Java user interface for Rodin that is provided as part of the plug-in

The transformation that is performed depends on the choice a user makes. If the user desires to create a visualization for each view, the transformation as shown in Figure 6.8 is performed. If the user wants to combine all views in a single visualization, the transformation as shown in Figure 6.9 is executed.

Both transformations map intermediate objects to BMS objects and intermediate attributes to BMS attributes. The key difference is that the second transformation combines all objects in one large visualization. The first transformation is best suited for large logical actions that must be analyzed. The second transformation can be used to analyze how multiple logical actions influence each other.

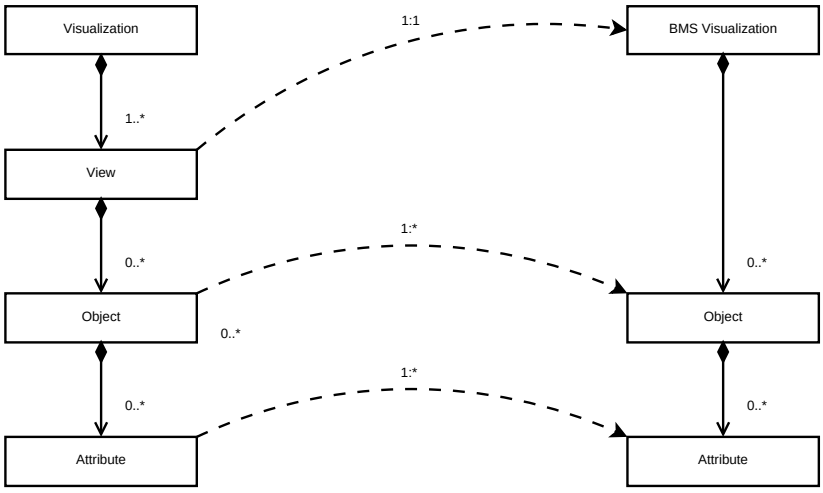


Figure 6.8: A first possible Java mapping from an intermediate visualization model to a BMS visualization

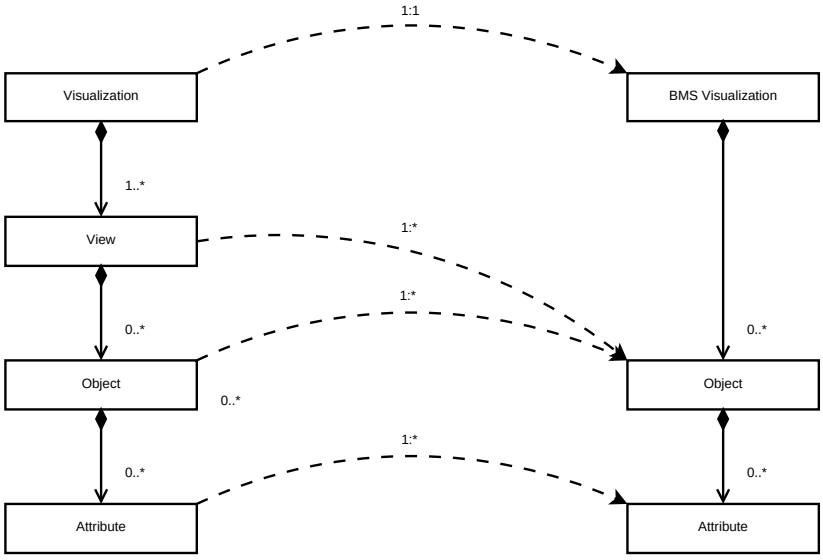


Figure 6.9: A second possible Java mapping from an intermediate visualization model to a BMS visualization

6.3.2 Transformation

The Java transformation is created in such a way that it uses the same structure as a QVTo transformation. The transformation starts with transforming the root node, for which it transforms the children. As all objects are a direct child of a view, the transformation has a limited depth.

Before the actual transformation can be performed, the intermediate model should be loaded as shown in Listing 6.8. The implementation uses an implementation that is commonly used to read XMI models. First, the intermediate package is initialized and factory for the visualization type is registered. Next, the file is loaded as resource using the registered resource factory. Finally, the resource is loaded and the root is casted to the desired type.

```
1  public IntermediateReader(String intermediateFile) {
2    this.file = intermediateFile;
3
4    // Initialize the model
5    IntermediatePackage.eINSTANCE.eClass();
6
7    // Register the XMI resource factory
8    Resource.Factory.Registry reg = Resource.Factory.Registry.INSTANCE;
9    Map<String, java.lang.Object> m = reg.getExtensionToFactoryMap();
10   m.put("visualization", new XMIResourceFactoryImpl());
11
12   // Obtain a new resource set
13   ResourceSet resSet = new ResourceSetImpl();
14
15   // Get the resource
16   Resource resource = resSet.getResource(URI
17     .createFileURI(file), true);
18
19   // Get the first model element and cast it to the right type, everything is included in
         this first node
20   intermediateVis = (asml.visualization.intermediate.Visualization) resource.getContents()
         .get(0);
21 }
```

Listing 6.8: The constructor of the IntermediateReader which loads the intermediate model

The model can be loaded for two purposes. On the one hand, the model can be loaded to list the views that are available during the import process. On the other hand, the model can be loaded when the actual transformation is performed. To perform the actual transformation, the function as presented in Listing 6.9 is invoked.The function creates a new BMS visualization using functionality that is already provided by the BMS plug-in. Second after which it starts the transformation of the root node. The function has the precondition that the `intermediateVis` variable contains a valid intermediate visualization root node.

```
1  public Visualization createVisualization(IFile file, String language, String version, List
       <Integer> views)  {
2
3    bmsVis = new Visualization(file.getName(), language, version);
4
5    convertVisualization(intermediateVis, views);
6
7    return bmsVis;
8  }
```

Listing 6.9: The root transformation function which creates a visualization from a set of views

```
1  private void convertVisualization(asml.visualization.intermediate.Visualization vis, List<
       Integer> views) {
2
3    // Start at [0, 0]
4    Point topLeft = IntermediateFactory.eINSTANCE.createPoint();
5
6    // Convert all given views
7    for (Integer viewIndex : views) {
8
9      // Convert the view
10     View view = vis.getViews().get(viewIndex);
11     convertView(view, topLeft);
12
13     // Update the top left coordinate to visualize all views next to each other
14     topLeft.setX(topLeft.getX() + view.getSize().getX() + 50);
15   }
16 }
```

Listing 6.10: The transformation function that visualized each view that is selected

To improve the readability and efficiency of the code, the created BMS visualization is accessed directly by all transformation functions. If the BMS visualization would only be updated in the top-level transformation, all functions would have to return their created objects which have to be saved.

The top-level level transformation is provided by the function as presented in Listing 6.10. It accepts the visualization that should be transformed in combination with a list of views that have to be combined in the BMS visualization. To combine multiple views, each view can be shifted independently. The current implementation distributes all views horizontally using the size information that is provided by the intermediate model. More elaborate placement algorithms can be implemented, but this simple algorithm performs well with the LACE models that are usually narrow.

Each view is transformed using the function as detailed in Listing 6.11. It performs the following operations.

1. A label is created that shows the name of the view in the top left corner.
2. All object of the view are gathered.
3. Objects are split into two groups: Connections and normal objects.
4. The normal objects are updated such that they contain a z-order attribute and they are shifted using the provided offset.
5. The group of normal objects is sorted using the z-order attribute such that objects are visualized starting with the lowest z-order value.
6. The group of connection objects is visualized after all normal objects are transformed. As connection objects in BMS are restricted to connecting existing objects, they need to be available during creation.

```
1  private void convertView(View view, Point shift) {
2
3      // Add a label that shows the name of the view
4      BText text = new BText(bmsVis);
5      text.setAttributeValue(AttributeConstants.ATTRIBUTE_TEXT, view.getName());
6      text.setAttributeValue(AttributeConstants.ATTRIBUTE_X, shift.getX() + 10);
7      text.setAttributeValue(AttributeConstants.ATTRIBUTE_Y, shift.getY() + 10);
8      text.setAttributeValue(AttributeConstants.ATTRIBUTE_WIDTH, 300);
9      text.setAttributeValue(AttributeConstants.ATTRIBUTE_HEIGHT, 20);
10     bmsVis.addChild(text);
11
12     // Get all objects of the view
13     EList<asml.visualization.intermediate.Object> objects = view.getObjects();
14
15     // Create two list for normal and connection objects
16     List<asml.visualization.intermediate.Object> normalObjects = new ArrayList<asml.
           visualization.intermediate.Object>();
17     List<asml.visualization.intermediate.Object> connectionObjects = new ArrayList<asml.
           visualization.intermediate.Object>();
18
19     for (Object object : objects) {
20       if (object.getType() == ObjectType.CONNECTION) {
21         // Add a connection to the set of connection objects
22         connectionObjects.add(object);
23       } else {
24
25         // Make sure that a z-order attribute is available (for sorting)
26         if (!object.getAttributes().containsKey("z-order")) {
27           Attribute attr = IntermediateFactory.eINSTANCE.createAttribute();
28           attr.setType(AttributeType.INTEGER);
29           attr.setValue(0);
30           object.getAttributes().put("z-order", attr);
31         }
32
33         // Shift the object horizontally and vertically
34         ...
35
36         normalObjects.add(object);
37       }
38     }
39
40
41     // Start with lowest z-value
42     Collections.sort(normalObjects, new Comparator<Object>() {
43       public int compare(Object object1, Object object2) {
44         return ((int)object1.getAttributes().get("z-order").getValue() - (int)object2.
               getAttributes().get("z-order").getValue());
45       }
46     });
47
48     for (Object object : normalObjects) {
49       convertNormalObject(object);
50     }
51
52     for (Object object : connectionObjects) {
53       convertConnectionObject(object);
54     }
55   }
```

Listing 6.11: The transformation function that creates a visualization of all objects of the given view

```
1  private void convertNormalObject(asml.visualization.intermediate.Object object) {
2    ObjectType type = object.getType();
3    EMap<String, Attribute> attributes = object.getAttributes();
4    BControl child = null;
5
6    switch (type) {
7      ...
8    case SHAPE:
9      String shape = (attributes.containsKey("shape") ? attributes.get("shape").getValue().
           toString() : "Rectangle");
10     child = new BShape(bmsVis);
11     child.setAttributeValue(AttributeConstants.ATTRIBUTE_ID, object.getId());
12     convertAttributes(object, child);
13     convertActions(object, child);
14     convertObservers(object, child);
15
16     if (shape.equals("Rectangle")) {
17       child.setAttributeValue(AttributeConstants.ATTRIBUTE_SHAPE, BAttributeShape.
             SHAPE_RECTANGLE);
18     } else if (shape.equals("Oval")) {
19       child.setAttributeValue(AttributeConstants.ATTRIBUTE_SHAPE, BAttributeShape.
             SHAPE_OVAL);
20     } else if (shape.equals("Triangle")) {
21       child.setAttributeValue(AttributeConstants.ATTRIBUTE_SHAPE, BAttributeShape.
             SHAPE_TRIANGLE);
22     } else if (shape.equals("Diamond")) {
23       child.setAttributeValue(AttributeConstants.ATTRIBUTE_SHAPE, BAttributeShape.
             SHAPE_DIAMOND);
24     }
25     bmsVis.addChild(child);
26
27     // Extra object to be able to set text for shapes
28     if (attributes.containsKey("text")) {
29       int shapeX = (int)attributes.get("x").getValue()+1;
30       int shapeY = (int)attributes.get("y").getValue()+1;
31       int shapeWidth = (int)attributes.get("width").getValue()-2;
32       int shapeHeight = (int)attributes.get("height").getValue()-2;
33       int textWidth = Math.min(shapeWidth, attributes.get("text").getValue().toString().
             length()*10);
34       int textHeight = Math.min(shapeHeight, 20);
35
36       BText textChild = new BText(bmsVis);
37       convertAttributes(object, textChild);
38       convertActions(object, textChild);
39       convertObservers(object, textChild);
40       textChild.setAttributeValue(AttributeConstants.ATTRIBUTE_ID, object.getId() + "_text
             ");
41       textChild.setAttributeValue(AttributeConstants.ATTRIBUTE_X, shapeX+(shapeWidth-
             textWidth)/2);
42       textChild.setAttributeValue(AttributeConstants.ATTRIBUTE_Y, shapeY+(shapeHeight-
             textHeight)/2);
43       textChild.setAttributeValue(AttributeConstants.ATTRIBUTE_WIDTH, textWidth);
44       textChild.setAttributeValue(AttributeConstants.ATTRIBUTE_HEIGHT, textHeight);
45       bmsVis.addChild(textChild);
46     }
47     break;
48     ...
49   }
50 }
```

Listing 6.12: The transformation function that converts normal objects of a view

Listing 6.12 shows one type of normal object (shape) that is transformed as part of the function that converts all normal objects. The shape transformation is the most representative case of the function and details some extra techniques that are used to convert attributes. The following steps are performed to achieve the desired functionality.

1. The type of the object is used to determine how the object should be transformed.
2. The shape attribute is added if it is missing.
3. The normal attributes, actions and observers are transformed using the transformation functions available.
4. The abnormal shape attribute is transformed manually as it cannot be transformed using the default approach.
5. The create shape object is added to the visualization.
6. As the shape objects of BMS do not support text, an additional label is created to display the text inside the shape.
7. The normal attributes, actions and observers are also used for the label as it should have the same functionality.
8. The information of the original shape is used to update the attributes of the text label such that it displays the text in the center of the shape.

Chapter 7

Results and Analysis

This chapter will provide an overview of the results achieved with the implementation. Furthermore, we analyze the implementation using a transformation validation and a users study.

7.1 Transformations

The implementation for the case study was created using a QVTo transformation, a Java transformation, and three new meta-models. The QVTo transformation has the following properties.

- 785 lines of code.
- 74 lines of comments.
- 1009 lines in total including whitespace.
- 5 `mapping` functions.
- 51 `helper` functions.
- 0 `query` functions.

The Java transformation statistics are as follows.

- 433 lines of code.
- 139 lines of comments.
- 648 lines in total including whitespace.
- 14 private functions.
- 5 public functions.

The high number of `helper` functions is caused by the high number of functions that are used to create attributes. Although attributes can be created in the same function that create the object, a clear separation improves the readability.

7.1.1 Technical Difficulties

The QVTo language has some limitations which must be overcome. Furthermore, the QVTo implementation used in the Borland 2008 IDE contains some bugs that introduce more problems. The following problems were encountered during the implementation of the QVTo transformation.

- The `EMap` created using the special construction as described in Section 6.1 has a problem saving ColorRGB values as part of an Attribute.
- Generated `EMap` variables cannot use sets as key.
- A set of sets introduces bugs when used directly to call mappings.
- The collect function provided by QVTo which can be used to collect a set of properties from a set of objects cannot be used for a set of sets.

Furthermore, the B-Motion Studio visualization framework has some limitations which had to be resolved.

First of all, not all shapes are supported. This was solved by using shapes that are closely related to the desired shape. If a more special shape is needed, the shape can be created using many small shapes, but this would probably have an impact on the performance of BMS.

Secondly, some shapes do not have the option to show text. By creating a label that is centered in the shape, as shown in Listing 6.12, this problem can be overcome.

Lastly, BMS cannot choose the proper event based on its guards. To be more specific, if one button can execute a set of events, only one can be executed in each state. To specify which of the events is executed, predicates can be used. If the predicate of one of the events holds, it will be chosen. However, if the guard of the event does not hold, the execution will fail and no action is executed at all.

A scan should be requested and executed using the same buttons used to request and execute normal subsystem actions. To accomplish this behavior in BMS, a different approach was used. The QVTo transformation creates multiple buttons in the same location, but only shows a button if the guards of the corresponding event hold. The user will not notice that multiple buttons are used because they look exactly the same.

The subsystem buttons are created by generating the power set of all subsystem names as shown in Listing 7.1. The first function collects the name of each subsystem. The second function is used to create the power set for given list of strings.

```
1  helper createSSPowerset(ssSet: OrderedSet(ActivityPartition)) : OrderedSet(Set(String)) {
2      var ssNames := ssSet->collect(represents.oclAsType(uml::Component).name)->asSet();
3      return powerSet(ssNames)->asOrderedSet();
4  }
5
6  helper powerSet(set: Set(String)) : Set(Set(String)) {
7      if (set->isEmpty()) then {
8          return Set{Set{}};
9      } endif;
10
11     var element := set->any(true);
12     var setMinElement := set->excluding(element);
13     var subSetPower := powerSet(setMinElement);
14     var subSetPowerWithElement : Set(Set(String)) := subSetPower->collectNested(including(
           element))->asSet();
15     return subSetPower->union(subSetPowerWithElement);
16 }
```

Listing 7.1: The helper that creates a power set of subsystem buttons

7.2 Visualized Models

Two logical action components were used for the validation of the transformations. The first LAC is an example LAC, created manually from scratch such that it contains all visual elements for which a transformation is created. The second LAC, developed by ASML, is used as validation and analysis of the simulation and transformation.

7.2.1 Example Model

The example LACE model as shown in Figure 7.1 has no meaningful behavior, but it contains all elements that should be transformed. For example, LA_One contains subsystems, subsystem actions, parallel execution, data flow and data transformation. Furthermore, LA_Three contains two scans for different subsystem combinations and another parallel execution block.

These three logical actions are used during development and early testing. If these are transformed correctly, more elaborate and meaningful logical action components can be used.

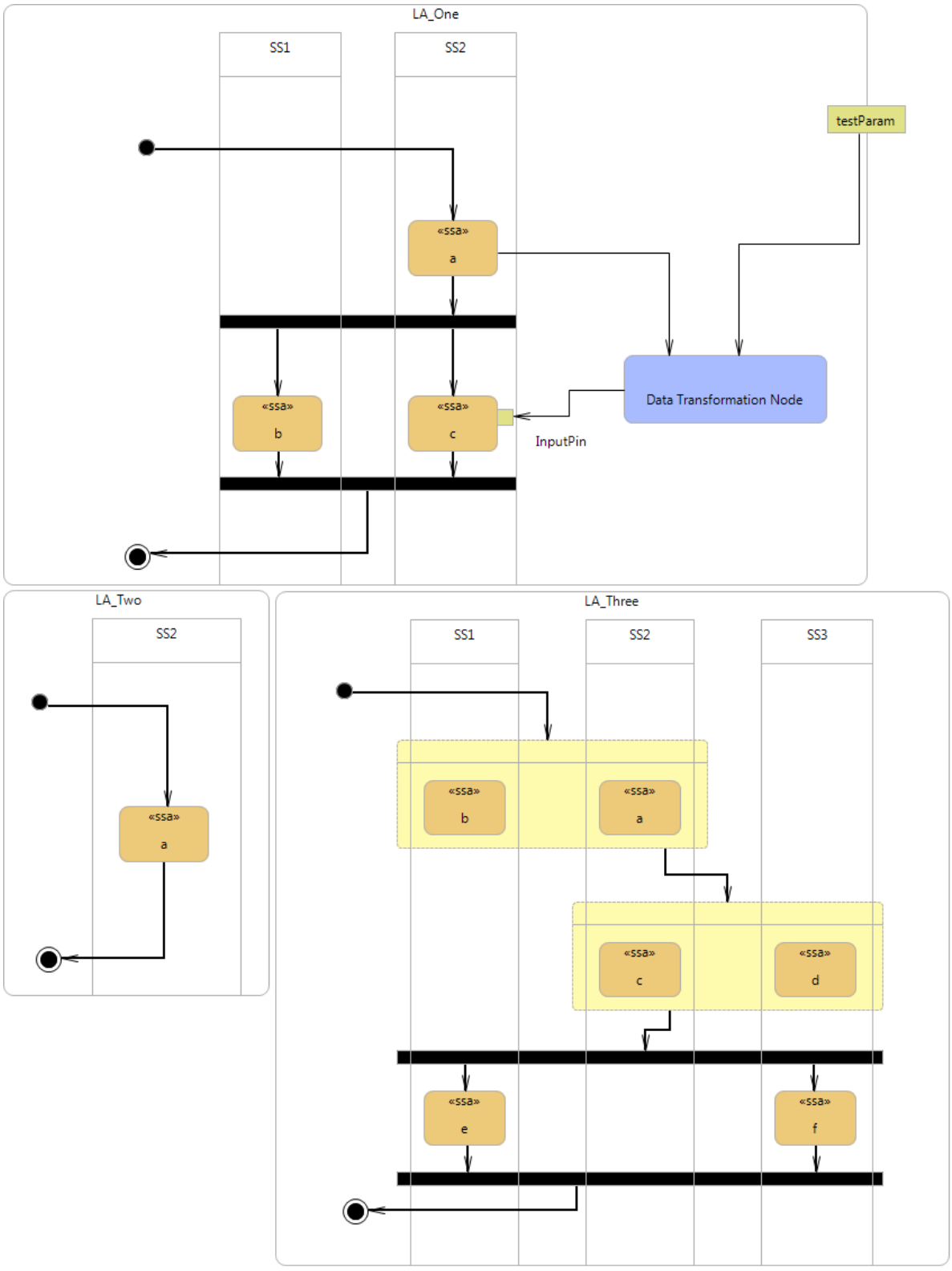


Figure 7.1: The three logical actions that are part of the example logical action

7.2.2 ASML Model

The second logical action component is visualized using the same approach. Although it has meaningful behavior for the developers, it uses only a subset of the functionality provided by LACE. For example, scans that only use all subsystems and no parallel execution.

7.3 Analysis results

The created transformation is analyzed in two ways. On the one hand, several existing LACE models are transformed to test if the created visualizations match the original LACE models. On the other hand, LACE users are invited to evaluate the visualization using a questionnaire. This also includes reviewing the behavioral aspects of the simulation.

7.3.1 Transformation Validation

The example model was transformed into an intermediate model of 238 kilobytes. The resulting visualization after importing the intermediate model in BMS is shown in Figure 7.2. This model was used to check the implementations during development. After several iterations, all logical actions could be transformed without a problem.

The visualization of the existing LACE model was obtained by importing an intermediate model of 498 kilobytes. Although multiple logical actions are available in the visualization, it is hard to view them all at once at their original size. The zoom functionality of BMS helps to get an overview of all logical actions in the LAC. Some logical actions use an odd location for the initial node resulting in a button that is placed outside the canvas.

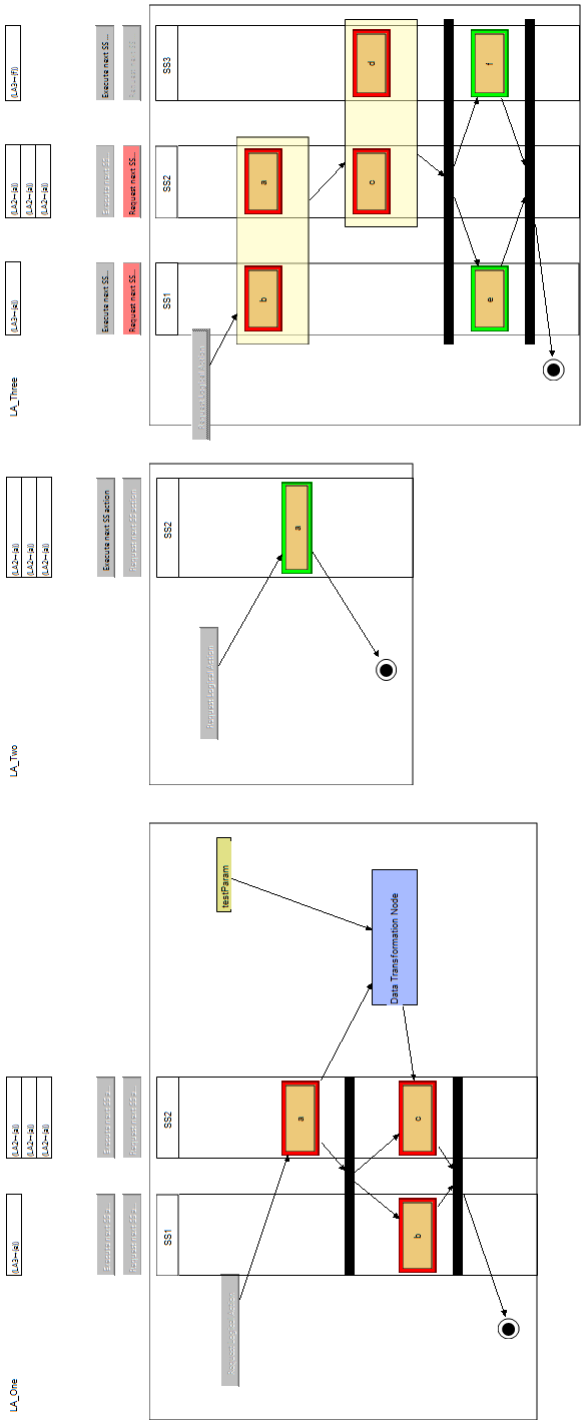


Figure 7.2: The visualization of the three logical actions that are part of the example logical action

7.3.2 Users Study

A users study was conducted to review the created visualization with LACE users. The results of the users study are also important for ASML. The questionnaire as shown in Appendix A was used to answer three main questions:

1. Do LACE users have problems with understanding how LACE works?
2. Would the simulated visualization help them to understand their LACE models?
3. Should ASML invest in such a visualization?

More than 20 LACE users were invited to review the visualization during a personal meeting. We met personally with ten ASML engineers, six of which use LACE. We demonstrated the visualizations of the LACE models as described in Section 7.2. After showing the visualizations and giving them the opportunity to use the visualization, we asked them to fill in the questionnaire as shown in Appendix A.

Four users did not have (much) LACE experience, but filled in the questionnaire as far as possible. One user reviewed the visualization with positive feedback, but did not fill in the questionnaire due to time limitations. ASML engineers located at Wilton, USA were also positive, but did not fill in the questionnaire.

Although not all users have LACE experience, the results as shown in Appendix B indicate that they understand the visualization quite well. To be more specific, 8 users indicate that the visualization would help understanding their LACE models. Furthermore, the results indicate that some LACE users understand the visualization better than the model as provided by the DSL environment. The visualization is also quite intuitive and with the right level of detail according to the user reviews.

The implemented features are appreciated, although many users would like to visualize existing execution traces. These traces can be extracted, modified and inspected using a part of the COREF project that is currently being developed by Maarten Manders as part of his PhD research. In addition, users indicate that they would appreciate an investment in this kind of visualization. If a mature visualization would be available, many users would use it during development, testing and simulation of existing execution traces. One user suggested that this type of visualization could also be useful to visualize the initialization sequence of a system.

ASML engineers were very positive in general. For example, user 7 wrote: "Good initiative, with the addition of playback it will be even more useful". Next, user 9 wrote: "Nice tool". Furthermore, ASML engineers were very enthusiastic about the visualization during the meetings.

Using the results of the questionnaire, we can answer our three main questions:

1. Although many users indicate that they understand their LACE models quite well, they also indicate that they had problems learning the behavioral aspects of LACE.
2. Yes, ASML engineers indicate that the visualization would help understanding their LACE models.
3. Yes, ASML should invest in this kind of visualization.

Chapter 8

Conclusions

Domain-specific languages are considered to be very effective in software development. By providing a high level of abstraction, developers can use domain concepts to implement functionality. However, debugging in the domain-specific environment is often performed by debugging the generated code.

To improve the usability of domain-specific languages, a graphical simulation of the execution of DSL models was introduced. While developing the implementation that generates this graphical simulation, many choices were made regarding the use of languages and tools. A list of requirements was set and used as guideline. Although existing solutions as described in Chapter 3 seemed promising, it became clear that a generic approach was necessary to meet all requirements.

The created implementation uses an intermediate meta-model to improve the reusability of the created model transformations. The first model transformation transforms the domain-specific models to a generic intermediate visualization model. The second model transformation creates a graphical simulation in the desired visualization framework.

The case-study was performed for the Logical Action Component Environment (LACE) DSL used within ASML. This is a domain-specific language which provides a graphical design interface to specify behavior on a high abstraction level. The graphical simulation uses the same layout as the original DSL model to improve the recognizability. A QVTo transformation was created to transform LACE models into intermediate models. Next, a Java implementation was created to transform the intermediate models into graphical simulation. The graphical simulation uses the B-Motion Studio plug-in of the Rodin framework.

The simulation is performed using an Event-B specification that is generated by the COREF project. The COREF project provides a framework that can be used to create formal specifications for domain-specific languages. The framework is meant for defining the dynamic semantics of DSLs and allows for mapping the DSL definition to the various platforms, such as verification, validation and simulation.

The implementation has been tested successfully in a realistic setup, using existing domain-specific models. At this point, we cannot yet provide evidence of how effective the new method is. According to the conducted users study, we expect it to be a valuable addition to the development and testing environment at ASML. It improves the usability of the DSL by making the behavior of models clear for domain-specific developers and by reducing the effort required to debug DSL models.

8.1 Research Goals

The section reflects on the research questions that were introduced in Section 1.1. Transformations can be optimized for reusability using similar techniques as used in normal software development. Creating separate sub-transformations or functions that can be reused in other transformations. This also includes the use of intermediate models that provide a stable basis for the input or

output of a transformation. An existing DSL van be enhanced using a graphical simulation by providing extra information to the end-user that is not available in the existing environment. By providing a visualization that resembles the original model, an end-user can use the visualization without a steep learning curve. This improves the usability of the DSL by providing a framework to debug and inspect the DSL model within an environment closely related to the DSL.

8.2 Future Work

To fully exploit the possibilities of the new approach, transformations for other domain-specific languages should be created. Furthermore, the current visualization framework (BMS) is deprecated and should be replaced with BMS 2 when available. The questionnaire also indicated that there is additional functionality, like the simulation of execution traces, desired by the domain-specific developers.

The ease of model transformation could be further improved by creating a graphical interface for the QVTo transformation. Another interesting research topic is the support for domain-specific languages that have no existing graphical representation. By creating a graphical simulation for these DSLs, more (text-based) domain-specific languages could be analyzed.

Bibliography

[1] U. Tikhonova, "A Framework for Defining the Dynamic Semantics of DSLs," in *Doctoral Symposium of 9th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2013)* (B. Meyer, L. Baresi, and M. Mezini, eds.), pp. 735–738, August 2013. 1, 10

[2] K. Hammond and G. Michaelson, "Hume: a domain-specific language for real-time embedded systems," in *Generative Programming and Component Engineering*, pp. 37–56, Springer, 2003. 5

[3] Z. Protic, *Configuration management for models: Generic methods for model comparison and model co-evolution*. PhD thesis, Eindhoven University of Technology, Eindhoven, The Netherlands, 2011. 6

[4] E. Consortium *et al.*, "Eclipse Graphical Modeling Framework (GMF)(2007)." 6

[5] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks, *EMF: Eclipse Modeling Framework*. Boston, MA: Addison-Wesley, 2 ed., 2009. 6

[6] J. Rumbaugh, I. Jacobson, and G. Booch, *Unified Modeling Language Reference Manual, The (2nd Edition)*. Pearson Higher Education, 2004. 7

[7] D. Méry and N. K. Singh, "A generic framework: from modeling to code," *Innovations in systems and software engineering*, vol. 7, no. 4, pp. 227–235, 2011. 9

[8] N. K. Singh, "EB2ALL: An automatic code generation tool," in *Using Event-B for Critical Device Software Systems*, pp. 105–141, Springer, 2013. 9

[9] D. Méry, N. K. Singh, *et al.*, "EB2C: A tool for Event-B to C conversion support," in *8th IEEE International Conference on Software Engineering and Formal Methods (SEFM)*, 2010. 9

[10] C. Snook and M. Butler, "UML-B: Formal modeling and design aided by UML," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 15, no. 1, pp. 92–122, 2006. 9

[11] T. C. de Sousa, C. F. Snook, and P. S. M. Silva, "A proposal for extending UML-B to support a conceptual model," *Innovations in Systems and Software Engineering*, vol. 7, no. 4, pp. 293–301, 2011. 9

[12] W. Hohmann, "Supporting model-based development with unambiguous specifications, formal verification and correct-by-construction embedded software," in *Proc. of SAE World Congress, Detroit, MI 2004*, 2004. 9

[13] A. Bouali and B. Dion, "Formal verification for model-based development," *SAE transactions*, vol. 114, no. 7, pp. 171–181, 2005. 9

[14] F. A. M. Do Nascimento, M. F. da Silva Oliveira, and F. R. Wagner, "Formal verification for embedded systems design based on MDE," in *Analysis, Architectures and Modelling of Embedded Systems*, pp. 159–170, Springer, 2009. 9

[15] M. Müller and D. Pfahl, "Simulation methods," in *Guide to Advanced Empirical Software Engineering*, pp. 117–152, Springer, 2008. 9

[16] B. Huang, G. Wu, L. Wan, L. Li, and J. Wang, "A visualization method of requirement checking based on software behavior," *Wuhan University Journal of Natural Sciences*, vol. 16, no. 6, pp. 507–512, 2011. 9

[17] H. Goldsby, B. H. Cheng, S. Konrad, and S. Kamdoum, "A visualization framework for the modeling and formal analysis of high assurance systems," in *Model Driven Engineering Languages and Systems*, pp. 707–721, Springer, 2006. 10

[18] K. Czarnecki and S. Helsen, "Classification of model transformation approaches," in *Proceedings of the 2nd OOPSLA Workshop on Generative Techniques in the Context of the Model Driven Architecture*, vol. 45, pp. 1–17, 2003. 10

[19] "Common Reference Framework (COREF) project." `www.win.tue.nl/mdse/COREF`. 10

[20] J.-R. Abrial, M. K. Lee, D. Neilson, P. Scharbach, and I. H. Sørensen, "The b-method," in *VDM'91 Formal Software Development Methods*, pp. 398–405, Springer, 1991. 14

[21] I. Ait-Sadoune and Y. Ait-Ameur, "Animating Event-B models by formal data models," in *Leveraging Applications of Formal Methods, Verification and Validation*, pp. 37–55, Springer, 2009. 16

[22] D. Méry and N. K. Singh, "Real-time animation for formal specification," in *Complex systems design & management*, pp. 49–60, Springer, 2010. 16

[23] J. Bendisposto and M. Leuschel, "A generic flash-based animation engine for ProB," in *B 2007: Formal Specification and Development in B*, pp. 266–269, Springer, 2006. 17

[24] L. Ladenberger, J. Bendisposto, and M. Leuschel, "Visualising Event-B models with B-Motion Studio," in *Formal Methods for Industrial Critical Systems*, pp. 202–204, Springer, 2009. 17

[25] I. Kurtev, "State of the art of QVT: A model transformation language standard," in *Applications of Graph Transformations with Industrial Relevance*, pp. 377–393, Springer, 2008. 18

[26] F. Allilaire, J. Bézivin, F. Jouault, and I. Kurtev, "ATL-eclipse support for model transformation," in *Proceedings of the Eclipse Technology eXchange workshop (eTX) at the ECOOP 2006 Conference, Nantes, France*, vol. 66, Citeseer, 2006. 18

[27] R. Dvorak, "Model transformation with operational QVT," *Borland Software Corporation*, 2008. 19

[28] M. Philippsen and B. Haumacher, "More efficient object serialization," in *Parallel and Distributed Processing*, pp. 718–732, Springer, 1999. 26

Appendix A

Questionnaire

Not at all — **Completely**

Do you know how your original LACE model works? O O O O O
Do you understand the visualization of your LACE model? O O O O O
Does your visualized LACE model behave as expected? O O O O O

Not intuitive at all — **Very intuitive**

Is the visualization intuitive to find the desired information? O O O O O
Is the visualization intuitive to execute the desired actions? O O O O O

Too general — **Too detailed**

Are you happy with the level of detail? O O O O O

How helpful are the following existing features? — **Not helpful** — **Very helpful** — **Depends on the usage (please explain)**

Colored SS actions to indicate queued and blocked actions O O O O O O _____
Colored buttons to indicate scans O O O O O O _____
Queue of subsystem actions that need to be executed O O O O O O _____
LA name in queues to relate to the queued actions O O O O O O _____

How helpful would the following additional features be?

List/Log of all requested logical actions O O O O O O _____
List/Log of processed SS actions for each SS O O O O O O _____
Data values for data flow O O O O O O _____
Simulation/Visualization of existing execution tracing O O O O O O _____

Not at all — **Very often**

Did you have problems learning LACE? O O O O O
Would you have learned LACE faster with this visualization? O O O O O

Do you still have problems understanding some lace models? O O O O O
Would this visualization help understanding those models? O O O O O

Would this kind of visualization save you time? O O O O O Time/Percentage? _____
Do you think ASML should invest in this kind of visualization? O O O O O _____

Figure A.1: The first part of the questionnaire used to review the created visualizations

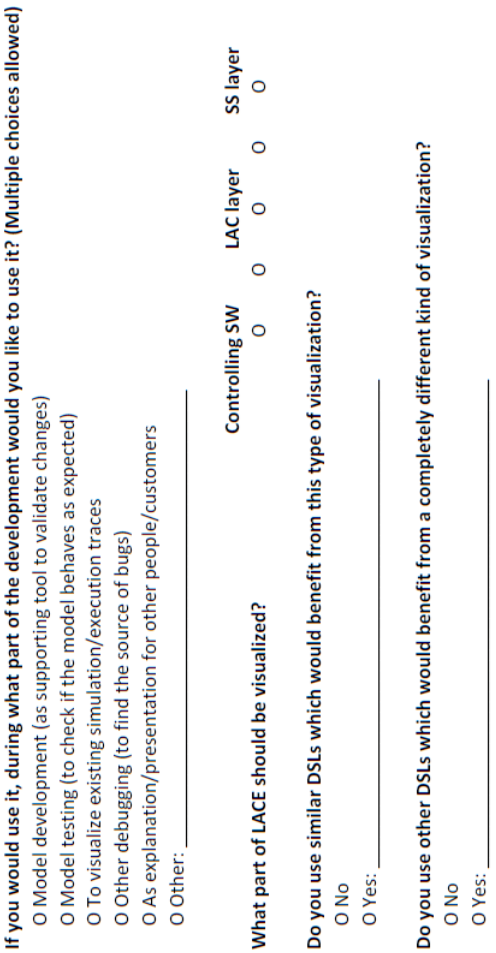


Figure A.2: The second part of the questionnaire used to review the created visualizations

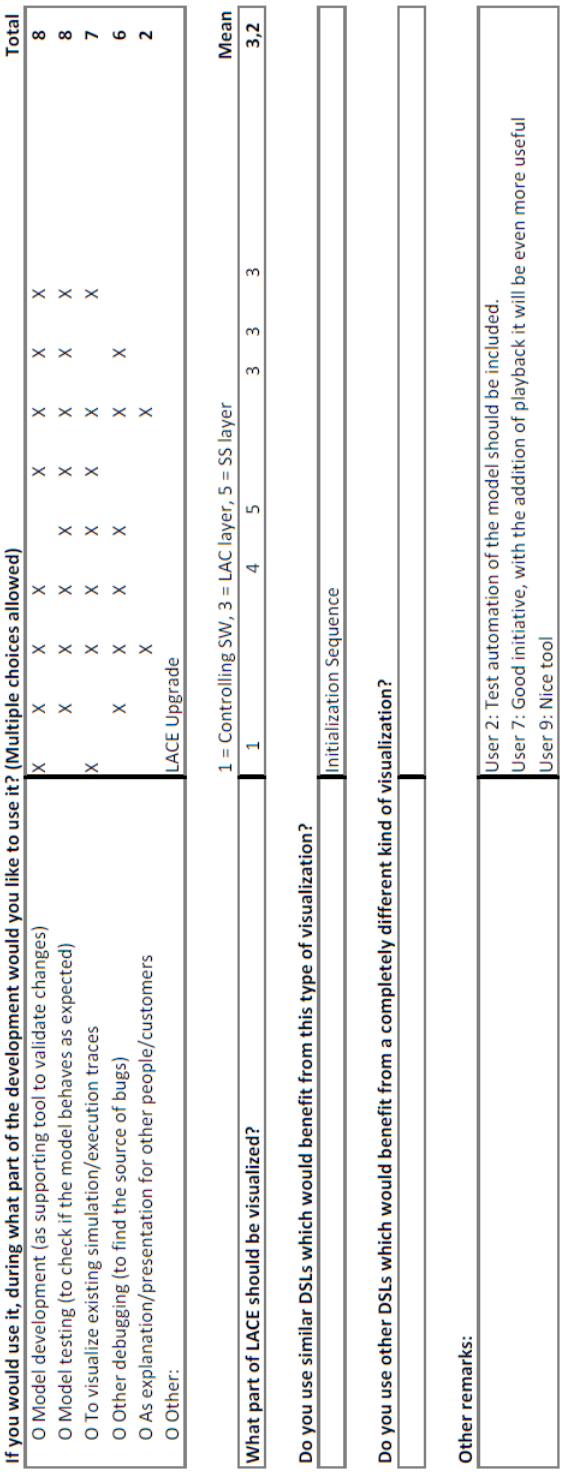
update

Appendix B

Questionnaire Results

User:	1	2	3	4	5	6	7	8	9		Mean
1 = Not at all, 5 = Completely											
Do you know how your original LACE model works?	3	2	3	3	5		5	4	5		**3,8**
Do you understand the visualization of your LACE model?	5	4	4	3	5	4	5	4	5		**4,3**
Does your visualized LACE model behave as expected?	5	3	4	3	4		5	4	5		**4,1**
1 = Not intuitive at all, 5 = Very intuitive											
Is the visualization intuitive to find the desired information?	4	4	4	4	3	4	4	4	4		**3,9**
Is the visualization intuitive to execute the desired actions?	4	4	4	4	3	4	4	4	4		**3,9**
1 = Too general, 3 = Yes, 5 = Too detailed											
Are you happy with the level of detail?	2	3	3	3	3	3	2	3	3		**2,8**
How helpful are the following existing features?	1 = Not helpful, 5 = Very helpful										
Colored SS actions to indicate queued and blocked actions	5	4	4	3	3	3	5	5	4		**4**
Colored buttons to indicate scans	4	4	4	3	3	3	4	5	4		**3,8**
Queue of subsystem actions that need to be executed	5	4	4	3	3	3	5	5	4		**4**
LA name in queues to relate to the queued actions	5	4	4	3	3	3	5	5	4		**4**
How helpful would the following additional features be?	1 = Not helpful, 5 = Very helpful										
List/Log of all requested logical actions	2	4	4	3	4		4	4	2		**3,4**
List/Log of processed SS actions for each SS	2	4	4	3	4		3	4	5		**3,6**
Data values for data flow	2	4	3	2	4		3	3	4		**3,1**
Simulation/Visualization of existing execution tracing	5	5	5	4	4		5	3	5		**4,5**
1 = Not at all, 5 = Very often											
Did you have problems learning LACE?	4			3	5		5	4	3		**4**
Would you have learned LACE faster with this visualization?	2			4	2		3	4	3		**3**
Do you still have problems understanding some lace models?	5			2	3		3	3	2		**3**
Would this visualization help understanding those models?	3	5		3	4	4	3	5	4		**3,9**
Would this kind of visualization save you time?		4	4	4	3		4	4	3	15/20%	**3,7**
Do you think ASML should invest in this kind of visualization?	4	5	4	4	4	4	4	5	4		**4,2**

Figure B.1: The first part of the questionnaire results



If you would use it, during what part of the development would you like to use it? (Multiple choices allowed)											Total
O Model development (as supporting tool to validate changes)	X	X	X	X		X	X	X	X		8
O Model testing (to check if the model behaves as expected)		X	X	X	X	X	X	X	X		8
O To visualize existing simulation/execution traces	X		X	X	X	X	X		X		7
O Other debugging (to find the source of bugs)		X	X	X	X		X	X			6
O As explanation/presentation for other people/customers			X				X				2
O Other:	LACE Upgrade										

	1 = Controlling SW, 3 = LAC layer, 5 = SS layer									Mean
What part of LACE should be visualized?	1			4	5		3	3	3	3,2

Do you use similar DSLs which would benefit from this type of visualization?
Initialization Sequence

Do you use other DSLs which would benefit from a completely different kind of visualization?

Other remarks:
User 2: Test automation of the model should be included. User 7: Good initiative, with the addition of playback it will be even more useful User 9: Nice tool

Figure B.2: The second part of the questionnaire results