Industrial Experiences with a Formal DSL Semantics to Check the Correctness of Generated DSL Artifacts*

Sarmen Keshishzadeh
Eindhoven University of Technology
Eindhoven, The Netherlands
s.keshishzadeh@tue.nl

Arjan J. Mooij
Embedded Systems Innovation by TNO
Eindhoven, The Netherlands
arjan.mooij@tno.nl

Jozef Hooman
Embedded Systems Innovation by TNO
Eindhoven, The Netherlands
Radboud University Nijmegen
Nijmegen, The Netherlands
jozef.hooman@tno.nl

A domain specific language (DSL) abstracts from implementation details and is aligned with the way domain experts reason about a software component. The development of DSLs is usually centered around a grammar and transformations that generate implementation code or analysis models. The semantics of the language is often defined implicitly and in terms of a transformation to implementation code. In the presence of multiple transformations from the DSL, the correctness of the generated artifacts with respect to the semantics of the DSL is a relevant issue. We show that a formal semantics is essential for checking the correctness of the generated artifacts. We exploit the formal semantics in an industrial project and use formal techniques based on equivalence checking and model-based testing for validating the correctness of the generated artifacts. We report about our experience with this approach in an industrial development project.

1 Introduction

A domain specific language (DSL) [8] abstracts from implementation details and is aligned with the way domain experts reason about a software component. By focusing on the essential concepts in a problem domain, DSLs facilitate the involvement of domain experts in the development of DSL specifications.

Tool support for the development of DSLs is improving constantly. Language workbenches such as Xtext enable language designers to define their languages and develop transformations that generate various artifacts from DSL models. This has boosted the popularity of DSL approaches in industry, as witnessed by reports like [16, 21].

The development of DSLs is usually centered around a grammar and transformations that generate implementation code or analysis models from DSL specifications. In such a setting, the main focus is on the transformation to implementation code which is very valuable in industrial practice. Generating analysis models is particularly interesting for safety-critical components and facilitates analyzing DSL models using a combination of formal techniques. For example, properties can be verified against verification models or simulation models can be used to explore the modeled behavior interactively.

In DSL approaches, the semantics of the language is often defined implicitly and in terms of the generated implementation. Although DSLs focus on the essential concepts of their respective domains,

*This research was supported by the Dutch national COMMIT program under the Allegio project, and by the European ARTEMIS program under the Crystal project.

J. Kofroň, J. Tumova, B. Buhnova (Eds.): Formal Engineering Approaches to Software Components and Architectures (FESCA'16)
EPTCS 205, 2016, pp. 16–30, doi:10.4204/EPTCS.205.2

the semantics of the language constructs are not always obvious. The lack of a formal semantics can give rise to different interpretations and cause inconsistencies between the transformations.

Various authors [1, 6] have proposed to use a formal semantics to describe the precise meaning of the language constructs. The formalization also allows them to have a single reference that should be followed by all the transformations. However, having a formal semantics as a reference does not guarantee the correctness of the transformations from a DSL. The developer of a transformation should have a deep understanding of the DSL and the target language and construct a transformation that does not deviate from the semantics of the DSL. In [7] the authors indicate that such tasks are very error-prone and introducing redundant validations is an effective way to reduce the rate of mistakes.

We propose to use the formal semantics of a DSL and introduce redundancy to validate the correctness of the artifacts generated from transformations [12]. We introduce redundant validations using the following formal techniques:

- equivalence checking (by transforming models to a formalism that allows checking equivalence of behaviors);
- model-based testing (by testing conformance of executable models to a test model).

We report on our experiences with these formal techniques in the context of an industrial DSL. This DSL is used in a development project to write specifications for an existing implementation of an industrial software component and to develop new enhanced specifications for future releases of the software. In this paper, we focus on DSL models that describe the existing implementation. The results obtained in the project show that the redundancy introduced by equivalence checking and model-based testing can effectively detect inconsistencies between the generated artifacts for a DSL. To hide the complexities of these techniques, we have developed a push-button technology that allows industrial users to automatically perform these checks for the generated artifacts.

Instead of validating the correctness of the generated artifacts, some authors [9] propose to formally prove the correctness of transformations. For a realistic DSL, this can be very costly in terms of time and the required expertise [14]. Thus, proving transformations should only be considered for well-established languages and transformations. Moreover, for a young DSL, transformations are improved regularly, and hence proving their correctness may not be effective. The industrial context of our work also calls for a pragmatic approach where there is no time for time-consuming proofs.

Overview. We discuss the mCRL2 process algebra in Section 2. In Section 3 we describe an industrial control component and informally introduce a DSL for describing its behavior; the semantics of the language is formalized in Section 4 using mCRL2. In Section 5 we use model-based testing to assess the quality of an implementation of the industrial component. The correctness of the used models is validated in Section 6. In Section 7 we discuss about our experiences with different types of model transformations. In Section 8 we discuss related work. Section 9 contains conclusions and directions for future research.

2 Preliminaries

In this section we give an overview of the micro Common Representation Language 2 (mCRL2) [11]. mCRL2 is a process algebra that extends the Algebra of Communicating Processes (ACP) [5] with data and time. The mCRL2 language and its supporting toolset [15] can be used to specify and analyze the behavior of distributed systems and protocols.

In this short overview, we focus on the language constructs that we need throughout the paper. The interested reader can refer to [11] for more details. We explain the way data types are defined and used in mCRL2 (Section 2.1) and describe behavioral specifications in the language (Section 2.2).

2.1 Data Specification

mCRL2 offers ways to specify data types (also known as sorts) and use their elements in specifications. Standard data types such as natural numbers ($\mathbb{N}$) and booleans ($\mathbb{B}$) are predefined in the language. Common operations on these data types are also available, e.g., $\approx$ denotes equality.

The user can define new data types in a specification. A new sort can be declared by explicitly characterizing its elements in a structured data type. For instance, we can define *Color* with elements *Red*, *Green*, and *Blue*:

sort $Color = \textbf{struct}\ Red \mid Green \mid Blue;$

It is also possible to declare structured types that depend on other sorts. For instance, a data type called *Message* that contains pairs of natural numbers can be defined as follows:

sort $Message = \textbf{struct}\ Pair(fst{:}\mathbb{N}, snd{:}\mathbb{N});$

A *Message* has the shape $Pair(n_1,n_2)$ where $n_1,n_2 \in \mathbb{N}$. The declaration of *Message* provides two projection functions, *fst* and *snd*, that extract the elements of $Pair(n_1,n_2)$:

$$fst(Pair(n_1,n_2)) = n_1 \qquad\qquad snd(Pair(n_1,n_2)) = n_2$$

Functions can also be declared and used in the mCRL2 language. Given two sorts A and B, the notation $A \to B$ denotes the sort of functions from A to B. mCRL2 includes an operator called function update for unary functions. For $f \in A \to B$ this operation is denoted by $f[a \to b]$ and represents a function that maps a to b and maps all the other elements of A like f does.

We can declare the sort of functions from $\mathbb{N}$ to $\mathbb{N}$ as follows:

sort $NatFunc = \mathbb{N} \to \mathbb{N};$

We consider two examples of this sort:

- *succ*: given $n \in \mathbb{N}$ returns $n+1$;
- *condsqr*: given $n \in \mathbb{N}$ returns n^2 if $n > 10$; otherwise, it returns n.

We declare these functions using the **map** keyword:

map $succ, condsqr : NatFunc;$

To define *succ* and *condsqr*, it is necessary to specify the calculations performed in these functions. This is realized by introducing equations using the keyword **eqn**. Variables used in the equations are declared by the keyword **var**.

var $n : \mathbb{N};$
eqn $succ(n) = n+1;$
$condsqr(n) = if(n > 10, n^2, n);$

The conditional operation has the shape $if(c,t,u)$. It evaluates to the term t if the condition c holds and it evaluates to the term u if c does not hold.

Lists are a built-in data type in mCRL2. The set of lists where all elements are from a sort A are represented by $List(A)$. Elements of $List(A)$ are built with two constructors: $[]$ the empty list, and $a \triangleright \ell$ which puts a (of type A) in front of list ℓ (of type $List(A)$). A list can be defined by specifying its elements and putting them between square brackets. For example, $[2,2,4]$ is a list of natural numbers.

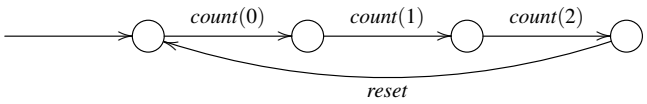


Figure 1: Behavior of *Counter*

2.2 Process Specification

mCRL2 allows us to specify behavior using a small set of primitives and operators. We use a simple example to describe some basic constructs of mCRL2. The example is a modulo 3 counter that starts counting from 0 and resets itself when it reaches 3.

In mCRL2, behavior is described in terms of processes. Actions are elementary processes and represent observable atomic events. Actions can also carry data parameters. In our example, *count* and *reset* can be considered as actions performed by the counter. The action *count* carries a data parameter to indicate the current number. These actions are declared as follows:

act $count : \mathbb{N}$;
$reset$;

Actions can be combined using different operators to form processes that specify more complex behaviors. For instance, the non-deterministic choice between process p and q is denoted by $p+q$ and the sequential composition of p and q is denoted by $p.q$. Data values can also influence the course of actions. Suppose c is a boolean expression. The process $c \to p \diamond q$ behaves as p if c holds and otherwise it behaves as q. The "else" part of the conditional operator can be omitted. If c does not hold in $c \to p$, deadlock will occur.

The following process specifies the modulo 3 counter. The process *Counter* carries a data parameter to keep track of the current number. The initial behavior is specified by **init**, i.e., counting starts from 0.

proc $Counter(n{:}\mathbb{N}) = (n < 3) \to count(n).Counter(n+1)$
$+ (n \approx 3) \to reset.Counter(0)$;

init $Counter(0)$;

For any $n \leq 3$ exactly one of the conditions $n < 3$ and $n \approx 3$ will evaluate to *true*. The process performs action $count(n)$ when $n < 3$ and then behaves as $Counter(n+1)$. If $n \approx 3$, the process performs *reset* and starts counting from 0. Fig. 1 depicts the labeled transition system of the counter.

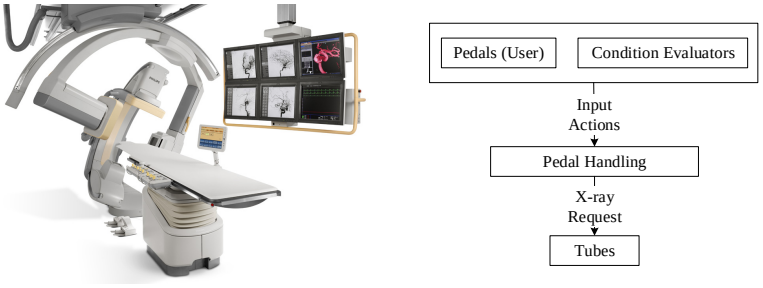
3 Industrial Application: a Clinical X-ray Generator

In this section we introduce an industrial control component that we use for reporting our experiences (Section 3.1). We also informally describe a DSL for specifying the behavior of the component (Section 3.2).

3.1 Platform

Philips Healthcare produces interventional X-ray systems (Fig. 2(a)) which are used to perform minimally-invasive medical procedures. During a procedure, the surgeon uses the images on the monitors as guidance. Images are constructed for two projections. The X-ray system consists of two planes: frontal (top-down) and lateral (left-right). These planes can be used separately or together (biplane).

The surgeon sends X-ray requests using the pedals. The interventional system of Fig. 2(a) includes a component called *Pedal Handling* that makes decisions about the amount of X-ray that should be



(a) Interventional X-ray (b) Interfaces of *Pedal Handling*

Figure 2: Industrial Application

generated in the tube of each plane (Fig. 2(b)). From each plane the following types of X-ray can be generated:

- Fluoroscopy: low dose X-ray, for obtaining real-time images;
- SingleShot: high dose X-ray, for capturing a single image;
- Series: high dose X-ray, for recording a series of images.

Pedal Handling also takes into account conditions that should interrupt the X-ray, or that should prevent the X-ray from starting. *Condition Evaluators* continuously evaluate these conditions and notify *Pedal Handling* when changes occur.

3.2 DSL for Pedal Handling

To describe the behavior of *Pedal Handling*, domain experts mainly focus on the external interfaces of the component. Starting from the initial state, they think about the input actions received from *Pedals* and *Condition Evaluators*. Based on the received input, the component might change its current state; it also makes a decision about the output X-ray and sends a request to the tubes. This process continues by receiving the next input. Thus, from the domain expert's point of view the behavior of *Pedal Handling* can be described with alternating sequences of input and output actions.

To specify the behavior of *Pedal Handling*, we use a DSL that fits this way of reasoning. Fig. 3 depicts a specification in the DSL. For confidentiality reasons, we do not provide a realistic model. This language is designed in collaboration with domain experts from Philips Healthcare. Although we use Fig. 3 as a running example to illustrate our approach, the results reported in the paper are based on realistic DSL models and implementations that are executable on the physical hardware.

A DSL model starts with declaring the input actions that can be received by the *Pedal Handling* component (`InActions`). Afterwards, it declares variables that keep track of the current state of the component (`Boolean variables` and `Plane variables`) and their initial values (`Init`). Two not-explicitly-declared variables, `OutputType` and `OutputPlane`, determine the output of *Pedal Handling*.

The internal logic of the component is described in terms of `Rules`. Each rule refers to an input action and consists of a guard and a do clause. The guard describes when the input action is enabled and the do clause determines how the action influences the state of the component and the output. A DSL model specifies exactly one rule for each input action. Multiple rules for an action are not supported.

Rules of the DSL use simple constructs for describing behavior. However, the precise meaning of some constructs is not obvious. For instance, it is not obvious whether evaluating the do clause of

```
InActions: FRFluoOn FRFluoOff
           StartCond ResetStartCond

Boolean variables: FRFluoReq FRFluoOK
Plane variables: FluoPlane

Init: FRFluoReq := false ;
      FRFluoOK := true ;
      FluoPlane := NONE ;

Rule: FRFluoOn guard: FRFluoReq == false
    do: FRFluoReq := true ;
        FluoPlane := FR ;
        if FRFluoOK
            then OutputType := Fluo ; OutputPlane := FR ;
        fi

Rule: FRFluoOff guard: FRFluoReq == true
    do: FRFluoReq := false ;
        if FluoPlane == FR
            then FluoPlane := NONE ;
            if OutputType == Fluo
                then OutputType := Standby ; OutputPlane := NONE ;
            fi
        fi

Rule: StartCond guard: FRFluoOK == true
    do: FRFluoOK := false ;

Rule: ResetStartCond guard: FRFluoOK == false
    do: FRFluoOK := true ;
```

Figure 3: Snapshot of a DSL Model

an action can be interrupted by receiving a new input action. Since a do clause may contain multiple assignments to a variable, it is also relevant to determine when a variable assignment takes effect.

In Section 4 we give a formalization of the DSL which clearly specifies the semantics of the language. For instance, our semantics enforces that do clauses should be interpreted in an atomic way and each assignment to a variable immediately changes its value such that the previous value is overwritten.

4 Formalizing the DSL Semantics

In Section 3.2 we informally introduced a DSL and motivated the need for a formal semantics. In this section, we give a formalization of the DSL semantics by introducing a transformation from the DSL to mCRL2. Similar to the *Pedal Handling* DSL, a compact representation/language for describing behavior in terms of labeled transition systems is very common in many application domains. Hence, our approach for formalizing the semantics of the DSL can be applied to other DSLs.

We describe our general transformation scheme by transforming the model of Fig. 3 to mCRL2. Our choice of mCRL2 is motivated by the expressiveness of the language, the availability of a toolset [15] that supports analysis of behavior, and our previous experience with the language and toolset. We first discuss the required data specifications (Section 4.1). Then we use process expressions to describe the behavior of DSL models (Section 4.2). Finally, we discuss the types of analysis that we perform on DSL models (Section 4.3).

4.1 Data Specification

Plane, X-Ray Type. As mentioned in Section 3.1, X-ray can be generated from three planes (frontal, lateral, and biplane). We define the data types *Plane* and *XRay* to describe the planes and the type of

X-ray generated from them.

sort $Plane = \textbf{struct}\ \ None \mid FR \mid LT \mid BI;$
$XRay = \textbf{struct}\ \ Standby \mid Fluo \mid SingleShot \mid Series;$

The combination of *None* and *Standby* describes a situation in which no X-ray is generated from the planes.

State. A DSL model declares a set of boolean and plane variables. The valuation of these variables and the two special variables `OutputType` and `OutputPlane` determine the state of the transition system described by the DSL model.

To describe the notion of state in mCRL2, we create two structured data types B (for boolean variables) and P (for plane variables). The structure of these sorts corresponds to the variable declarations of the DSL model. For the model of Fig. 3, B and P are defined as follows:

sort $B = \textbf{struct}\ FRFluoReq \mid FRFluoOK;$
$P = \textbf{struct}\ FluoPlane;$

To specify the valuations of boolean and plane variables, we declare the following data types:

sort $BVals = B \to \mathbb{B};$
$PVals = P \to Plane;$

Finally, the notion of state can be formalized as follows:

sort $PSt = \textbf{struct}\ St(bs{:}BVals, ps{:}PVals, outType{:}XRay, outPlane{:}Plane);$

The projection functions *outType* and *outPlane* extract the values of `OutputType` and `OutputPlane` from states.

In a DSL model, the initial values of the boolean and plane variables are specified by `Init`. We declare $bs_0 \in BVals$ and $ps_0 \in PVals$ to specify the initial values in mCRL2. We also declare $s_0 \in PSt$ to describe the initial state.

map $bs_0 : BVals;$
$ps_0 : PVals;$
$s_0 : PSt;$
eqn $bs_0(FRFluoReq) = false;$
$bs_0(FRFluoOK) = true;$
$ps_0(FluoPlane) = None;$
$s_0 = St(bs_0, ps_0, Standby, None);$

The initial values of `OutputType` and `OutputPlane` cannot be specified by `Init` in the DSL. It is assumed that initially no X-ray is generated from the planes. Hence, we specify s_0 such that:

$$outType(s_0) = Standby \qquad outPlane(s_0) = None$$

Guard, Do Clause. A DSL model specifies one rule for each input action. The rule of an action consists of a guard and a do clause. From Fig. 3 one can see that a guard is a function from states to booleans. A do clause consists of a sequence of assignments/conditionals that given the current state can change the values of the variables and produce a new state. We describe guards and do clauses as follows:

sort $Guard = PSt \to \mathbb{B};$
$DCl = List(PSt \to PSt);$

For the DSL model of Fig. 3 with 4 rules, we declare the guards and do clauses g_i, d_i for $1 \leq i \leq 4$:

map $g_1, g_2, g_3, g_4 : Guard$;
$d_1, d_2, d_3, d_4 : DCl$;

The calculations of each guard can be described in terms of an equation. For instance, the guard of the first rule of Fig. 3 can be defined as follows:

var $b : BVals$;
$p : PVals$;
$xr : XRay$;
$pl : Plane$;
eqn $g_1(St(b,p,xr,pl)) = (b(FRFluoReq) \approx false)$;

To describe do clauses, we specify assignments and conditionals in terms of equations. To explain this, we consider the do clause of the first rule from Fig. 3. This do clause contains four assignments and one conditional.

We describe each assignment as a function that updates one of the components of a state argument $St(b,p,xr,pl)$. We denote the assignments of the first rule by a_1, a_2, a_3, a_4 based on their order of appearance:

map $a_1, a_2, a_3, a_4 : PSt \to PSt$;
eqn $a_1(St(b,p,xr,pl)) = St(b[FRFluoReq \to true], p, xr, pl)$;
$a_2(St(b,p,xr,pl)) = St(b, p[FluoPlane \to FR], xr, pl)$;
$a_3(St(b,p,xr,pl)) = St(b,p,Fluo,pl)$;
$a_4(St(b,p,xr,pl)) = St(b,p,xr,FR)$;

For example, the first assignment updates the values of boolean variables by setting *FRFluoReq* to *true*.

A conditional statement of a do clause is specified by a term of the shape $if(c,t,u)$ in equations. The conditional in the first rule of Fig. 3 can be described as follows:

map $cond : PSt \to PSt$;
eqn $cond(St(b,p,xr,pl)) = if(b(FRFluoOK), Eval(condthen, St(b,p,xr,pl))$
$, St(b,p,xr,pl))$;

In this equation, *Eval* is a function that evaluates a sequence of assignments or conditionals and *condthen* is the "then" part of the conditional (see below).

A do clause is described as a sequence of assignments/conditionals. For example, we describe the do clause of the first rule (d_1) as a sequence of the assignments a_1, a_2 and the conditional *cond*. The "then" part of a conditional is also specified as a sequence of its components. The "then" part of the conditional in the first rule (*condthen*) is a sequence of a_3, a_4:

map $condthen : List(PSt \to PSt)$;
eqn $d_1 = [a_1, a_2, cond]$;
$condthen = [a_3, a_4]$;

Having sequences of assignments and conditionals, it is also essential to define a function that applies a sequence of statements to a state and returns the resulting state. The following mCRL2 description defines *Eval* for this purpose:

map $Eval : List(PSt \to PSt) \times PSt \to PSt$;
var $s : PSt$;
$f : PSt \to PSt$;
$\ell : List(PSt \to PSt)$;
eqn $Eval([], s) = s$;
$Eval(f \triangleright \ell, s) = Eval(\ell, f(s))$;

The function *Eval* is defined by specifying its effect on terms of the shape $[]$ and $f \triangleright \ell$ (the constructors of *List*).

4.2 Process Specification

The *Pedal Handling* component performs two types of actions: input and output. A DSL model explicitly declares a set of input actions (from *Pedals* and *Condition Evaluators*) by `InActions`. *Pedal Handling* also performs actions $output(xr, p)$ to send requests for $xr \in XRay$ to $p \in Plane$. This action is not explicitly declared in the DSL. Corresponding to the input actions and the output action we declare actions in mCRL2. For Fig. 3 we declare:

act $FRFluoOn, FRFluoOff, StartCond, ResetStartCond;$
$output : XRay \times Plane;$

In Section 3.2, we mentioned that domain experts describe the behavior of *Pedal Handling* with alternating sequences of input and output actions. The semantics of the DSL is aligned with this intuition. We specify the semantics of the DSL model of Fig. 3 in terms of the following processes:

proc $P_{In}(s{:}PSt) = g_1(s) \rightarrow (FRFluoOn.P_{Out}(Eval(d_1, s)))$
$+ g_2(s) \rightarrow (FRFluoOff.P_{Out}(Eval(d_2, s)))$
$+ g_3(s) \rightarrow (StartCond.P_{Out}(Eval(d_3, s)))$
$+ g_4(s) \rightarrow (ResetStartCond.P_{Out}(Eval(d_4, s)));$
$P_{Out}(s{:}PSt) = output(outType(s), outPlane(s)).P_{In}(s);$
init $P_{In}(s_0);$

The process P_{In} describes the behavior of *Pedal Handling* when the component is ready to receive an input. It carries a data parameter that indicates the current state. The process P_{In} uses a combination of choices and conditionals for case distinction. The guards are used as conditions in the conditional operators to determine enabled actions. Performing an input action updates the state based on the corresponding do clause. The process P_{Out} describes the behavior of *Pedal Handling* when the component is ready to produce an output. In this situation, *output* is performed and it carries the X-ray type and plane extracted from the state. Performing the output action does not influence the state.

The processes P_{In} and P_{Out} enforce alternating execution of the input and output actions. Do clauses are evaluated by *Eval*. Thus, new actions cannot be performed before a do clause is completely evaluated. Moreover, assignments have an immediate effect.

4.3 Analyzing DSL Models

For a safety critical component, it is desired to use DSL models as a single source to automatically obtain models that enable analysis using various formal techniques, e.g., verification, simulation.

To enable verification, we have automated the transformation from the DSL to mCRL2. We have used the mCRL2 toolset to generate the state spaces of DSL models and to verify properties expressed in a variant of the modal μ-calculus. A realistic DSL model declares 25 input actions and their effects in terms of rules. The corresponding state space consists of approximately 45000 states and 350000 transitions. We have verified some safety properties against this model, e.g., “deadlock-freedom”, “no X-ray is generated from the planes when there is no request from the user”. The interested reader can refer to Appendix A of [13] for a modal μ-calculus formalization of some safety properties for the DSL model of Fig. 3.

The mCRL2 formalization can also be used as a reference to implement transformations to other formalisms. Having a formalized semantics helps to avoid arbitrary choices in the transformations that would give a different semantics to the constructs of the DSL.

To enable simulation, we have implemented an automated transformation from the DSL to POOSL [19, 17]. POOSL is a modeling language with a semantics expressed in terms of timed probabilistic

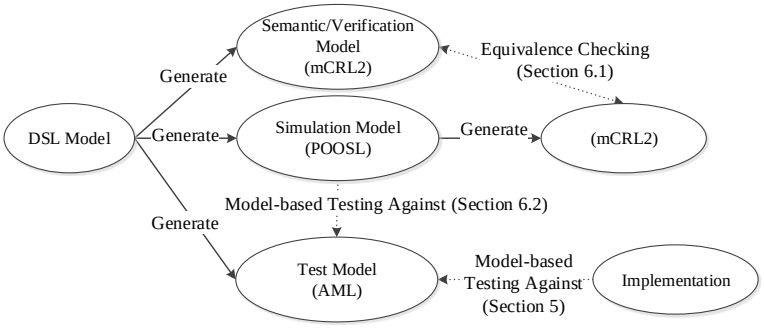


Figure 4: Generated Artifacts for *Pedal Handling* and their Validation

labeled transition systems. The tools available for POOSL allow us to simulate the modeled behavior and discuss our observations with domain experts. Due to space restrictions, we do not discuss this transformation; the interested reader can refer to [13] for a detailed description of the transformation.

Using the mCRL2 formalization as guidance for implementing a transformation to POOSL does not give a robust connection between the generated mCRL2 and POOSL models. In Section 6, we formally validate the correctness of POOSL models with respect to the mCRL2 formalization. A proposed implementation for *Pedal Handling* is available. Thus, at the moment we do not generate code from the DSL. Fig. 4 depicts the transformations from the DSL; test models are discussed in Section 5. In Section 6 we also discuss an approach for validating simulation models against test models.

5 Validating the Industrial Implementation

We introduce model-based testing as a way to assess the correctness of an implementation with respect to a DSL description (Section 5.1). We also discuss about interpreting the results obtained from model-based testing (Section 5.2).

5.1 Validating the Implementation by Model-based Testing

In industry, the implementation is considered to be the most valuable artifact that is produced for a component. When behavioral models of a component are also available, it is relevant to check whether the implementation complies to the modeled behavior. Validating the compliance of implementations to DSL models adds a level of redundancy that can reveal discrepancies between the developed DSL models and implementations.

We use model-based testing to validate the correctness of an implementation with respect to a DSL model. Model-based testing uses a model that describes the behavior of a system under test and enables both automatic generation and execution of test cases on the implementation.

The semantics of the *Pedal Handling* DSL is described in terms of labeled transition systems and hence we create a test environment based on the theory of input-output conformance (*ioco*) testing for labeled transition systems [20] to automatically derive test cases from the behavior described in the DSL and execute them on the implementation.

In the *ioco* theory, correctness of implementations with respect to specifications is expressed in terms of the binary relation of *ioco*. The *ioco* theory provides an algorithm that derives a set of test cases from

a given specification such that executing this set of test cases on an implementation determines whether the specification and implementation are related by the conformance relation.

The model-based testing tool of Axini [3] is based on the *ioco* theory. In this tool, specifications are described in the Axini Modeling Language (AML). We have developed an automated transformation from the DSL to AML (Fig. 4). The generated AML model for a DSL specification is used for model-based testing against the implementation; see [13] for details about the transformation to AML.

5.2 Interpreting the Results of Model-based Testing

Validating the compliance of an implementation to a DSL model by model-based testing requires special attention to interpret the results correctly. A failed test case shows a discrepancy between the test model and the implementation. This can have three different reasons:

- a mistake in the transformation from the DSL to the test models;
- a failure in the implementation;
- a modeling mistake in the DSL model, which is cascaded to the test model.

The transformation from the DSL to test models should preserve the semantics of the DSL. A mistake in realizing the semantics may result in failed test cases. In Section 6 we introduce an approach to gain confidence in the correctness of the generated models in Fig. 4. When there is sufficient confidence in the correctness of test models, a failed test case may indicate a failure of the implementation.

The last item mentioned above is particularly relevant if implementations are not generated from the DSL. In such cases, failed test cases could also indicate a mistake in DSL models; the intended behavior is not correctly described in the DSL (and the same modeling mistakes are cascaded to the test model) but the implementation has correctly realized the behavior.

Results. For our model-based testing experiments, we used a real but not-yet-released implementation of *Pedal Handling*. Independently of our experiments with the DSL, the developers have constructed an extensive suite of unit tests for validating the implementation. However, in model-based testing the focus is on assessing the observable behavior of the component based on a model by providing stimuli at its external interfaces. This revealed a number of issues in the implementation that are out of the scope of the unit tests. For example, one of the requirements of *Pedal Handling* indicates that high-dose X-ray requests have priority over low-dose X-ray requests. A design mistake in realizing this requirement led to a failed test case. The failed test was part of an unlikely trace where three different pedals must be pressed at the same time.

Model-based testing revealed that certain modeling choices taken in our DSL models are implemented differently in the implementation. Unlike stopping *Fluo* (modeled by *FRFluoOff* in the example of Fig. 3), stopping *Series* requires performing two actions in a specific order. The output specified for the first step of stopping *Series* was different from the output produced by the implementation. This observation led to minor changes in our DSL models.

6 Checking the Correctness of Generated Models through Redundancy

Transformations from a DSL to analysis models allow the user to apply various formal techniques and reason about DSL models. Analysis models can also be used to assess the correctness of other artifacts

available for a component (Section 5). However, the results obtained from analysis models are only valuable if the corresponding transformations correctly realize the semantics of the DSL.

Developing a transformation from a DSL to a modeling language requires a deep understanding of the semantics of the DSL and the target language. Moreover, the transformation should not deviate from the semantics of the DSL. Introducing redundancy is a very effective way to reduce the rate of mistakes in such error-prone tasks [7].

We introduce redundancy to validate the correctness of the artifacts depicted in Fig. 4 in two ways: equivalence checking (Section 6.1) and model-based testing (Section 6.2). Note that the validation techniques introduced in this section are not bound to the transformations of Fig. 4 and hence manually constructed models can also be validated using equivalence checking and model-based testing.

6.1 Checking the Behavioral Equivalence between Artifacts

The verification and simulation models in Fig. 4 have an underlying labeled transition system. To get confidence in the correctness of simulation models with respect to the formalized semantics (transformation to verification models), we can investigate whether the labeled transition systems described by the simulation and verification models are related by an equivalence relation, e.g., strong bisimulation. This may require to develop transformations from analysis models to a formalism that enables state space generation and comparison.

We have used the mCRL2 toolset for state space generation (the `lps2lts` tool) and comparison (the `ltscompare` tool). To enable state space generation for simulation models, we have developed a transformation from POOSL to mCRL2. To avoid bridging wide semantic gaps between POOSL and mCRL2, we have restricted our simulation models to a sufficient subset of POOSL (see [13] for more details about the constructs used in simulation models) and developed a transformation to mCRL2 for that specific subset.

When comparing behaviors, the internal steps performed by them are not relevant; we focus on the observable behaviors. Hence, we check whether the state spaces are equivalent modulo branching bisimulation. Fig. 4 depicts equivalence checking between mCRL2 and POOSL models and the automated transformation from POOSL to mCRL2 that enables this check.

Results. For realistic DSL models, the behaviors described in mCRL2 and POOSL were equivalent modulo branching bisimulation. Models generated from three transformations (DSL to mCRL2, DSL to POOSL, and POOSL to mCRL2) are involved in equivalence checking between simulation and verification models. Each transformation is implemented by a different person. This reduces the probability of identical mistakes in the transformations and makes the redundancy introduced by equivalence checking more valuable.

6.2 Model-based Testing of Executable Models

In Section 5 we used model-based testing to validate the correctness of an implementation. Executable analysis models can also be treated as black-boxes that interact via their interfaces; we can supply inputs to an executable model and observe its output. Thus, executable models can be tested against the test model generated from a DSL model using model-based testing. Failed test cases reveal mistakes in the transformations. In our case study, we have applied model-based testing to POOSL models; see Fig. 4.

Note that there are also other possibilities for validating the artifacts of Fig. 4 (e.g., model-based testing mCRL2 models against AML models). Adding such validations gives more confidence in the

correctness of the artifacts but would require additional effort to create the required environment.

Results. We did not encounter any failed test cases in our model-based testing experiments against POOSL models. Similar to Section 6.1, the transformations to simulation models (POOSL) and test models (AML) are implemented by different people. Absence of failed test cases gives more confidence that the semantics of the DSL is realized correctly in these models.

7 Experiences with Different Kinds of Model Transformations

The approach of Fig. 4 deploys model transformations to enable the use of multiple formal techniques and to introduce redundant mechanisms for assessing the correctness of different artifacts with respect to the formalized semantics. This approach relies on two types of transformation: transformations from the DSL to a general-purpose modeling language (from the DSL to mCRL2, POOSL, and AML), and transformations between general-purpose modeling languages (from POOSL to mCRL2). In this section we report on our experiences with these two types of transformations.

General-purpose modeling languages originate from different disciplines and are applicable to a wide range of problems. For instance, POOSL is designed to be applicable for simulation and performance analysis, whereas mCRL2 is focused on formal verification. In our experience, making transformations between two general-purpose formalisms is not a trivial task. There are usually language constructs from the source language that are difficult or even impossible to translate to the target language.

For example, the data layer of POOSL is object-oriented. Each data class describes its variables and methods. Moreover, instances of certain data structures (e.g., strings, lists) are accessed using pointers. On the other hand, mCRL2 is not object-oriented and does not support pointers. Hence, transforming data classes or pointers from POOSL would require complex mechanisms in mCRL2 models.

Similarities between the source and target languages may suggest a direct mapping between certain constructs. However, similar constructs in two formalisms may have subtly different semantics. For instance, one would expect the conditional statement `if c then p else q fi` of POOSL to be trace equivalent to $c \rightarrow p \diamond q$ in mCRL2. Based on the formal semantics of POOSL, `if c then p else q fi` first performs an internal step (τ transition) to evaluate the condition. Then it behaves as `p` if `c` holds and otherwise it behaves as `q`. However, in mCRL2 no internal action is performed for evaluating c and hence the two conditional statements are not trace equivalent. This observation reiterates the importance of formal semantics in transformations between languages.

In the literature, some authors have reported similar experiences about transformations between general-purpose modeling languages. A partial transformation from the hybrid modeling formalism Chi 2.0 [4] to mCRL2 is proposed in [10]. The intention is to enable formal verification on models in Chi 2.0. Semantic differences between Chi 2.0 and mCRL2 makes the transformation complex and hard to maintain. The generated models are also complex and sometimes difficult to analyze by tools.

In our experience, restricting the scope of a transformation (e.g., using predefined data types in POOSL specifications instead of representing POOSL data classes in mCRL2) and studying the semantics of the relevant constructs from the source and target languages are effective ways to overcome the challenges faced in implementing transformations between general-purpose languages.

The *Pedal Handling* DSL is defined for a narrow domain and its semantics is less elaborate compared with POOSL, mCRL2, and AML. Thus, it requires less effort to construct the transformations from the DSL to different analysis models.

8 Related Work

In [1] the authors prototype the semantics of a DSL called SLCO in terms of a transformation to an intermediate language called CS. Afterwards, CS models are transformed to labeled transition systems and are inspected manually or analyzed by existing tools. The authors have also implemented a number of transformations from SLCO to SLCO models with equivalent behaviors and suggested that the prototype semantics can be used to compare the underlying labeled transition systems of the source and target models.

The B method has been used in [6] to develop process schedulers based on specifications in a DSL. The information given by a DSL model is taken into account at several refinement steps in B machines. The authors also introduce a decidable logic for expressing proof obligations of the refinement steps. This allows them to automatically prove the refinements.

The mentioned studies validate refinement steps, whereas we offer various formal techniques to assess the semantic correctness of different types of artifacts in an automated way.

In [18] a combination of model-based techniques is used to develop a software bus in a two-phase process. In the first phase, an mCRL2 model of the component is created and validated through simulation. After developing the component, the mCRL2 model is used for model-based testing of the implementation. In the second phase, different properties are verified against the mCRL2 model. The model is improved based on the results and is used for model-based testing against a second implementation. In comparison, our approach is centered around domain-specific models and artifacts generated from them.

9 Conclusions

A DSL allows us to use models that are naturally aligned with the way domain experts reason about a software component. Existing tools enable language designers to define DSLs and to construct transformations to implementation code and analysis models. However, the semantic correctness of the generated artifacts is usually overlooked.

To resolve conflicting interpretations of a DSL, we use a formal semantics of the language. We also propose to have additional mechanisms to validate the correctness of the generated artifacts with respect to the semantics of the DSL.

We have experimented with this approach as preparation for the redesign of a clinical X-ray generator. In this paper, we reported on our experiences with DSL models for an existing implementation of the industrial component. At the moment, the DSL and the transformation to simulation models are frequently used for discussions on potential enhancements in the behavior of the component.

We plan to extend our approach by other forms of redundancy to increase the reliability of the artifacts. To this end, we consider using model learning. Model learning techniques extract an automaton model for an implementation by systematically performing tests on it and observing its behavior [2]. A learned model can give insight in the implemented behavior, and can be compared with the labeled transition system described by a DSL model.

Our approach for validating the generated artifacts for a single DSL model can be extended to validate the transformations themselves based on a set that consists of several DSL models. Various criteria can also be defined for the considered sets to obtain DSL models that examine different aspects of the transformations.

References

[1] S. Andova, M.G.J. van den Brand & L. Engelen (2011): *Prototyping the Semantics of a DSL using ASF+SDF: Link to Formal Verification of DSL Models*. In: *Proceedings of AMMSE'11*, *EPTCS* 56, pp. 65–79, doi:10.4204/EPTCS.56.5.

[2] D. Angluin (1987): *Learning regular sets from queries and counterexamples*. *Information and computation* 75(2), pp. 87–106, doi:10.1016/0890-5401(87)90052-6.

[3] Axini: `http://www.axini.nl`.

[4] D.A. van Beek, A.T. Hofkamp, M.A. Reniers, J.E. Rooda & R.R.H. Schiffelers (2008): *Syntax and formal semantics of Chi 2.0*. *Eindhoven University of Technology, Technical Report*.

[5] J.A. Bergstra & J.W. Klop (1984): *Process algebra for synchronous communication*. *Information and control* 60(1), pp. 109–137, doi:10.1016/S0019-9958(84)80025-X.

[6] J.P. Bodeveix, M. Filali, J. Lawall & G. Muller (2005): *Formal methods meet domain specific languages*. In: *Proceedings of IFM'05*, Springer, pp. 187–206, doi:10.1007/11589976_12.

[7] M.G.J. van den Brand & J.F. Groote (2013): *Software Engineering: Redundancy is Key*. *Science of Computer Programming* 97, pp. 75–81, doi:10.1016/j.scico.2013.11.020.

[8] A. van Deursen, P. Klint & J. Visser (2000): *Domain-Specific Languages: an annotated bibliography*. *SIGPLAN Notices* 35(6), pp. 26–36, doi:10.1145/352029.352035.

[9] H. Ehrig & C. Ermel (2008): *Semantical Correctness and Completeness of Model Transformations Using Graph and Rule Transformation*. In: *Proceedings of ICGT'08*, *LNCS* 5214, Springer-Verlag, pp. 194–210, doi:10.1007/978-3-540-87405-8_14.

[10] Stappers F.P.M. (2012): *Bridging Formal Models: An Engineering Perspective*. Ph.D. thesis, Eindhoven University of Techonology.

[11] J.F. Groote & M.R. Mousavi (2014): *Modeling and Analysis of Communicating Systems*. MIT press.

[12] S. Keshishzadeh & A.J. Mooij (2016): *Formalizing and Testing the Consistency of DSL Transformations*. *Formal Aspects of Computing (in press)*, doi:10.1007/s00165-016-0359-1.

[13] S. Keshishzadeh, A.J. Mooij & J. Hooman (2015): *Industrial Experiences with a Formal DSL Semantics to Check Correctness of DSL Transformations*. *arXiv preprint:1511.08049*.

[14] X. Leroy (2009): *Formal verification of a realistic compiler*. *Communications of the ACM* 52(7), pp. 107–115, doi:10.1145/1538788.1538814.

[15] mCRL2: `http://mcrl2.org`.

[16] A.J. Mooij, J. Hooman & R. Albers (2013): *Gaining Industrial Confidence for the Introduction of Domain-Specific Languages*. In: *Proceedings of IEESD'13*, IEEE, pp. 662–667, doi:10.1109/COMPSACW.2013.83.

[17] POOSL: `http://poosl.esi.nl`.

[18] M. Sijtema, A. Belinfante, M.I.A. Stoelinga & L. Marinelli (2014): *Experiences with formal engineering: Model-based specification, implementation and testing of a software bus at Neopost*. *Science of computer programming* 80, pp. 188–209, doi:10.1016/j.scico.2013.04.009.

[19] B.D. Theelen, O. Florescu, M.C.W. Geilen, J. Huang, P.H.A. van der Putten & J.P.M. Voeten (2007): *Software/Hardware Engineering with the Parallel Object-Oriented Specification Language*. In: *Proceedings of MEMOCODE'07*, IEEE, pp. 139–148, doi:10.1109/MEMCOD.2007.371231.

[20] J. Tretmans (2008): *Model based testing with Labelled Transition Systems*. In: *Formal methods and testing*, *LNCS* 4949, Springer, pp. 1–38, doi:10.1007/978-3-540-78917-8_1.

[21] J. Verriet, H.L. Liang, R. Hamberg & B. van Wijngaarden (2013): *Model-driven development of logistic systems using domain-specific tooling*. In: *Proceedings of CSD&M*, Springer, pp. 165–176, doi:10.1007/978-3-642-34404-6_11.