An Exercise in Iterative Domain-Specific Language Design*

Marcel van Amstel Mark van den Brand Luc Engelen
Eindhoven University of Technology
P.O. Box 513, 5600 MB
Eindhoven, The Netherlands
{M.F.v.Amstel | M.G.J.v.d.Brand | L.J.P.Engelen}@tue.nl

ABSTRACT

We describe our experiences with the process of designing a domain-specific language (DSL) and corresponding model transformations. The simultaneous development of the language and the transformations has lead to an iterative evolution of the DSL. We identified four main influences on the evolution of our DSL: the problem domain, the target platforms, model quality, and model transformation quality.

Our DSL is aimed at modeling the structure and behavior of distributed communicating systems. Simultaneously with the development of our DSL, we implemented three model transformations to different formalisms: one for simulation, one for execution, and one for verification. Transformations to each of these formalisms were implemented one at the time, while preserving the validity of the existing ones. The DSL and the formalisms for simulation, execution, and verification have different semantic characteristics. We also implemented a number of model transformations that bridge the semantic gaps between our DSL and each of the three formalisms. In this paper, we describe our development process and how the aforementioned influences have caused our DSL to evolve.

1. INTRODUCTION

Domain-specific languages (DSL) and model transformations are the key concepts in model driven engineering [22]. A DSL is a language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain [31]. A DSL enables software engineers to model using domain concepts rather than concepts provided by existing formalisms, which typically do not provide the required or correct abstractions. Models created using a DSL can be transformed into models in other formalisms using model transformations. Model transformations can for example be used to transform models specified using a DSL into models that are suitable for applying formal methods to. In this way, formal methods can be applied without having to create separate models for that purpose [21, 29]. In this article, we will show how to use model transformation to apply formal methods to models specified using our DSL.

The literature provides guidelines for developing DSLs [18, 30, 31], and tools that support DSL evolution [8, 23]. There have been numerous reports on evolution of DSLs. The changes to the SDF language, a DSL used for syntax definition, are described by Visser [32]. Van Beek et al. describe the evolution of Chi, a language for modeling and simulating hybrid systems [28]. The evolution of a language used for interchanging models of hybrid systems is described by Van Beek et al. [27]. In most of these publications, only the changes themselves are described and the reasons for these changes are only hinted upon. On the underlying reasons for DSL evolution literature is scarce and conclusions are scattered. Also, most analyses of DSL evolution provide an a posteriori report only. In this paper, we discuss our experiences with an evolving DSL during its iterative design.

We designed a DSL for modeling systems consisting of concurrent, communicating objects. The structure of a system is modeled using classes and their behavior is modeled by state machines. Simultaneously to the development of the DSL, we implemented a number of model transformation to different formalisms: one for simulation, one for execution, and one for verification. These model transformations were developed consecutively. First, a model transformation was implemented to enable simulation of the models using POOSL [25]. In this way, models developed using an intuitive, graphical syntax can be simulated without the need for modelers to learn the syntax and semantics of a simulation language. Second, a model transformation was implemented to enable execution of the models on the Lego Mindstorms platform [2]. Executing the code generated from a model revealed bugs in the model that originated from unforseen interleavings of concurrent objects. These bugs were not encountered during simulation. To detect this kind of problems, a third model transformation was implemented to enable verification of the models using Spin [13]. All three of the aforementioned formalisms have semantic properties that are different from the semantic properties of

*This work has been carried out as part of the FALCON project under the responsibility of the Embedded Systems Institute with Vanderlande Industries as the industrial partner. This project is partially supported by the Netherlands Ministry of Economic Affairs under the Embedded Systems Institute (BSIK03021) program.
This work has been carried out as part of the KWR 09124 project LithoSysSL.

IWPSE-EVOL '10, September 20–21, 2010 Antwerp, Belgium

our DSL. To enable model transformations from our DSL to each of these platforms, the semantic gaps need to be bridged [26]. Therefore, we implemented model transformations that transform models specified in our DSL to different but equivalent models also specified in our DSL where the semantic properties are aligned with the required target platform. This approach has as advantage that the transformations to the different platforms only need to transform syntax. We describe the development process and indicate how our DSL has evolved during this process. Note that we focus on language evolution only and not on the co-evolution of models specified in the DSL such as described in [6]. Even though our DSL is small in terms of the provided number of modeling constructs, we expect that the lessons learned are applicable to projects involving larger DSLs as well. In our discussion of related work in Section 5 we show that the conclusions drawn by others overlap with our own.

The remainder of this article is structured as follows. Section 2 introduces our DSL and the languages for simulation, verification, and execution. In Section 3, we describe the development of our model transformations. Section 4 describes the evolution the language has undergone. Related work is discussed in Section 5. In Section 6 we draw some conclusions. Finally, Section 7 gives directions for further research.

2. INVOLVED LANGUAGES

Our goal is to have a DSL with an intuitive graphical syntax to model the structure and behavior of a system consisting of concurrent, communicating objects. Models specified using this DSL should be transformed into models for simulation, verification, and execution. To ensure that the same model is simulated, verified, and executed, we implemented model transformations from our DSL to languages for the respective purposes. In this way, the simulation, verification, and execution models do not have to be created by hand. This has as advantage that engineers do not have to learn the syntax and semantics of different languages. Moreover, manual transformation is a slow and error-prone task. In collaborations with industrial partners, we observe that they find formal methods useful, but hard to use. Using a transformational approach such as the one presented here, formal methods can be applied without having to know their details.

In the remainder of this section we discuss the constructs provided by our DSL and their informal semantics. We also introduce the languages used for simulation, verification, and execution. Note that the languages we have chosen are merely examples of languages for simulation, verification, and implementation. We could have chosen completely different languages to achieve the same goal.

2.1 Specification

We designed and implemented a DSL called *Simple Language of Communicating Objects* (SLCO). SLCO provides constructs for specifying systems consisting of objects that operate in parallel and communicate with each other. Figure 1 shows the main metaclasses and relations of the SLCO metamodel. The remainder of the metamodel is shown in Figure 2, which elaborates on the abstract metaclasses *Statement*, *Trigger*, and *Expression*.

An SLCO model consists of a number of classes, instances of these classes (objects), and channels. Channels are used

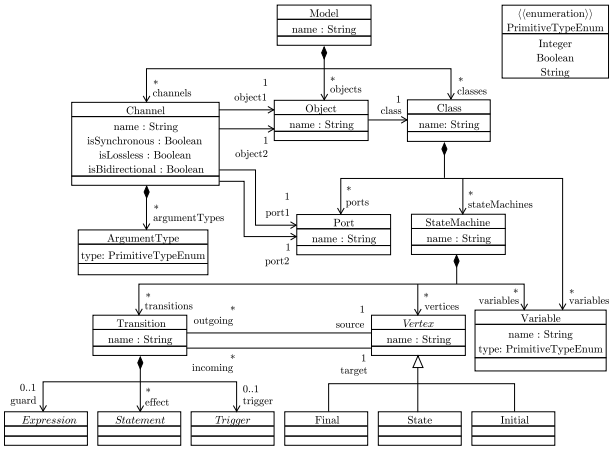
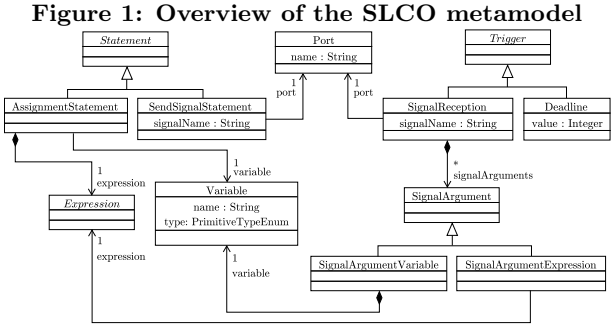
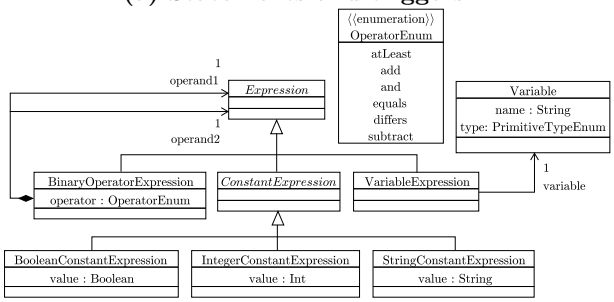


Figure 1: Overview of the SLCO metamodel



(a) Statements and triggers



(b) Expressions

Figure 2: Statements, triggers, and expressions in SLCO

to connect a pair of objects such that they can communicate with each other. An example of this is shown in Figure 3. The objects *Left*, *Middle*, and *Right*, which are instances of classes S and M, can communicate over channels *M2L* and *M2R*. A class describes the structure and behavior of its instances. A class has ports and variables that define the structure of its instances, and state machines that describe their behavior. Ports are used to connect channels to objects. Figure 3 shows that both instances of class S have a port *Middle* connecting them to channels *M2L* and *M2R*, and that the instance of class M has two ports, *Left* and *Right*, connecting it to the same channels. A state machine consists of variables, states, and transitions. A transition has a source and a target state, possibly a guard or a trigger, and a number of statements that form its effect. A guard is a boolean expression that must hold to enable the transition from source state to target state. There are

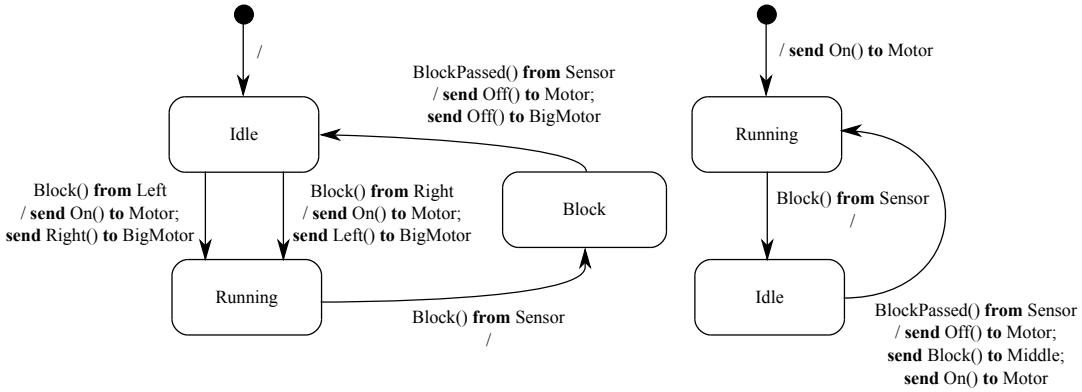


Figure 4: Two state machines in SLCO

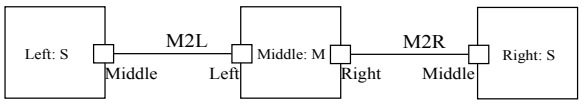


Figure 3: Three objects connected by channels in SLCO

two types of triggers: a deadline and a signal reception. If the amount of time specified by a deadline has passed or if a signal is received, the transition that has such a trigger is enabled. When a transition is made from one state into another state, the statements that constitute the effect of the transition are executed. There are statements for assigning values to variables and for sending signals over channels. Figure 4 shows an example of a state machine.

2.2 Simulation

We use the Parallel Object-Oriented Specification Language (POOSL) [25], a formal modeling language for simulation and performance analysis, for simulation of SLCO models. The behavioral part of POOSL is based on the formal language CCS [19] and the part for modeling data is based on traditional object-oriented languages. A POOSL model consists of a set of concurrent processes connected by channels. These processes can communicate by exchanging synchronous messages via these channels. POOSL is supported by two tools: SHESim and Rotalumis. POOSL models can be interactively simulated using the built-in POOSL interpreter of SHESim. Rotalumis is a command-line tool that can simulate POOSL models at high speed.

2.3 Verification

Model checking is an automated verification technique that checks whether a formally specified property holds for a model of a system [7]. We use the model checker Spin [13] for verifying our models. Spin can, among others, check a model for deadlocks, unreachable code, or determine whether it satisfies an LTL property [20]. Linear Temporal Logic (LTL) is used to express properties of paths in a finite-state representation of the state space of a system. The input language for Spin is Promela. The Promela language has constructs for modeling selections and loops, based on Dijkstra's guarded commands. Promela has primitives for message passing between processes over channels either using queues, or handshaking. This enables modeling of both asynchronous and synchronous communication, respectively. The syntax of expressions and assignments in the Promela language is similar to that of C.

2.4 Execution

To execute SLCO models, an implementation platform is required. We chose to use the Lego Mindstorms [2] platform for this purpose. The key part of this platform is a programmable Lego brick, called RCX. This RCX has an infrared port for communication and is connected by wires to sensors and motors for interaction with its environment. We deliberately opted for the outdated RCX brick instead of the newer and more advanced NXT brick to investigate the strength of our transformational approach when dealing with a primitive execution platform. The language we use to program these programmable bricks is called Not Quite C (NQC) [4]. NQC is a restricted version of C, combined with an API that provides access to the various capabilities of the Lego Mindstorms platform such as sensors, outputs, timers, and communication.

2.5 Language Characteristics

Each of the languages we discussed has different characteristics. These differences are shown in Table 1. The first column lists the languages. The second column indicates whether communication over channels is synchronous or asynchronous in the corresponding language. In case communication is synchronous, both sender and receiver need to be available before a signal can be sent. In this way, sender and receiver synchronize on communication. In case communication is asynchronous, a sender can send a signal and proceed with its execution even though the receiver is not yet ready to receive the signal. The third column lists the channel directionality. Channels can be either unidirectional or bidirectional. This means that channels can either be used for sending signals in one direction only or in both directions, respectively. Although Spin provides bidirectional channels, the asynchronous versions of these channels are actually buffers that make it possible for a process to receive signals that it sent itself. Therefore, we will use channels in Spin in one direction only. The fourth column indicates the channel reliability, i.e., whether a channel is lossless or lossy. In case a channel is lossless, a signal that is sent will always arrive at the receiving end. In case a channel is lossy, a signal that is sent may get lost. The fifth column lists the amount of objects that can be instantiated simultaneously.

	(A)synchronous channels	Channel directionality	Channel reliability	Concurrent objects	Number of channels
SLCO	both	Bidirectional	both	∞	∞
POOSL	synchronous	Bidirectional	lossless	∞	∞
Promela	both	Unidirectional	lossless	∞	∞
NQC	asynchronous	Bidirectional	lossy	limited	1

Table 1: Language characteristics

In SLCO, POOSL, and Spin this amount is unlimited. For Lego Mindstorms, however, this number is limited in practice. Because every object should be deployed on an RCX, the amount of concurrent objects is bounded by the available number of RCX bricks. The sixth column lists the amount of communication channels that are available in the different languages. This amount is unlimited for SLCO, POOSL, and Spin. The RCXs used on the Lego Mindstorms platform communicate using infrared. This means that all objects use the same medium for communication, hence there is only one channel.

3. DEVELOPMENT PROCESS

In the Section 2, we explained that the language characteristics of the various platforms differ. To enable simulation, verification, and execution, these semantic platform gaps need to be bridged [26]. Therefore, we defined a number of transformations that transform an SLCO model to a different SLCO model with equivalent observable behavior. An SLCO model that uses synchronous communication only, for example, can be transformed to an equivalent SLCO model that uses asynchronous communication only. We also defined transformations from SLCO to POOSL, Promela, and NQC. These transformations can only be performed if the characteristics of the SLCO model that is to be transformed match the characteristics of the target platform. To transform *any* SLCO model, the possible platform gaps should be bridged by first performing the necessary transformations to replace all the constructs used in the SLCO model that do not exist on the target platform. After this alignment of the platform characteristics of SLCO and the target language, the transformations to the different formalisms merely transform syntax. In the remainder of this section, the transformations that bridge platform gaps as well as the transformations to the different platforms will be explained in detail.

3.1 Bridging Platform Gaps

3.1.1 Synchronous Communication over Asynchronous Channels

In this transformation, synchronous signals are replaced by asynchronous signals. To ensure that the behavior of the model remains the same, acknowledgment signals are added for synchronization. Whenever a signal is sent, the receiving party sends an acknowledgement indicating that the signal has been received. The sending party waits until the acknowledgement has been received. In this way, synchronization is achieved.

3.1.2 Replacing Bidirectional Channels by Unidirectional Channels

To ensure that channels are used in one direction only, we defined a transformation that replaces bidirectional channels by a pair of unidirectional ones. This transformation also ensures that a signal is sent over the appropriate unidirectional channel such that it is used in one direction only. Suppose channel C between objects A and B is replaced by a channel C_{AB} from object A to object B and a channel C_{BA} from object B to object A. The transformation ensures that all signals sent by object A to object B will be sent via channel C_{AB}, and vice versa.

3.1.3 Lossless Communication over Lossy Channels

To ensure lossless communication over a lossy channel, the alternating bit protocol (ABP) presented in [3] is added for every class that communicates over a lossless channel in the source model. The protocol will ensure that a signal will arrive at its destination eventually, under the premise that not all signals get lost. In this way, lossless communication over a lossy channel is achieved.

3.1.4 Decreasing the Number of Objects

In practice, the number of concurrent objects is limited on the Lego Mindstorms platform. Therefore, we defined a transformation that merges objects to decrease their number. In SLCO, it is impossible for two state machines that belong to the same object to communicate over channels. When objects are merged, it may occur that state machines belonging to different objects are merged into the same object, preventing them from communicating over channels. In this case, communication over channels is replaced by passing data using shared variables, using an implementation of the four-phase hand-shake protocol [10] to ensure proper synchronization.

3.1.5 Decreasing the Number of Channels

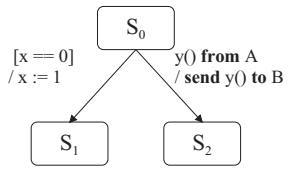
There is only one channel available on the Lego Mindstorms platform for communication between RCXs. Therefore, we defined a transformation that decreases the number of channels that is used for communication between objects. This transformation merges all channels between a pair of objects. When multiple channels are merged, the destination of signals is no longer unambiguous. Therefore, signals are tagged with an identifier to determine their destination.

3.2 Cross-Platform Transformations

3.2.1 SLCO to POOSL

The transformation from SLCO to POOSL transforms SLCO models containing synchronous, lossless channels to POOSL models. This transformation transforms each class to a process class. The state machines of each class are transformed to a number of process methods, one for each state. An example of part of the transformation is shown in Figure 5. Each process method representing a state contains a select statement consisting of a number of guarded statements representing the outgoing transitions of that state. If

one or more of these guarded statements are enabled, one of them is chosen non-deterministically and executed. If none of the statements are enabled, the select statement is blocked until one of the statements it contains becomes enabled. The guards of the statements represent the guards of outgoing transitions, and the statements represent the triggers and effects of those transitions. Each guarded group of statements ends with a process method call that calls the method representing the target state of the transition represented by this guarded group of statements.



(a) Partial state machine

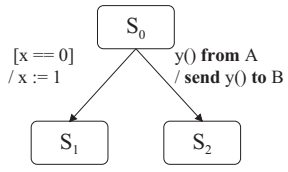
```
S0()()
 ||
 sel
  [x == 0]
  x := 1;
  S1()()
 or
  A?y();
  B!y();
  S2()()
 les
```

(b) Corresponding POOSL code

Figure 5: Transforming SLCO to POOSL

3.2.2 SLCO to Promela

The transformation from SLCO to Promela transforms SLCO models containing synchronous or asynchronous, lossless channels to Promela models. Every state machine contained in a class instantiated as an object in an SLCO model is transformed to a Promela proctype. Channels between objects are transformed to channels between proctypes. State machines can be implemented using an imperative programming style in multiple ways. We have chosen to implement them as jump tables using goto statements. An example of part of the transformation is shown in Figure 6. State S_0 depicted in Figure 3.2.2 is transformed to the code depicted in Figure 3.2.2. A state is transformed to a labeled selection



(a) Partial state machine

```
S0:
 if
 ::x == 0 -> x = 1;
   goto S1;
 ::A?y -> B!y
   goto S2;
 fi;
```

(b) Corresponding Promela code

Figure 6: Transforming SLCO to Promela

statement. Every outgoing transition of state S_0 is transformed to an alternative of the selection statement. The semantics of the selection statement is such that it will non-deterministically execute one of the alternatives for which the guard is executable and it will block if none of the guards is executable. The guard is the first statement or expression of the alternative. The guard and statements of a transition are transformed to Promela code in a straightforward way. When the guard is executable, the guard itself and the statements following it can be executed. In case the guard is an expression, it is executable if it evaluates to `true`. In case the guard is a statement, it is executable if the statement is. When the guard and the statements are executed, the transition to a state has been completed and the state machine is *in* the target state of the transition. Therefore, a goto statement that jumps to the label representing the target state of a transition is added after the transformed statements in the code. Because we use an untimed version of Spin, we abstract from time by transforming deadline triggers to skip statements. A signal reception trigger is transformed to a receive statement. A receive statement blocks until it is able to receive a message over a channel.

3.2.3 SLCO to NQC

The transformation from SLCO to NQC transforms SLCO models with asynchronous, lossy channels to NQC models. Every state machine contained in a class instantiated as an object in an SLCO model is transformed into an NQC task. The transformation from state machines to NQC code resembles the transformation from state machines to Promela described in Section 3.2.2. There is however an important difference. The semantics of the if statement in NQC is different from the one in Promela.

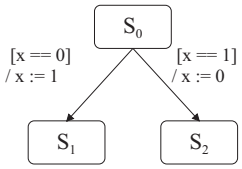
First, the if statement in NQC is a conditional statement rather than a selection construct. This means that only one alternative can be specified per if statement. Therefore, every outgoing transition of a state is transformed into a separate if statement.

Second, the if statement in NQC is non-blocking. This means that the condition of an if statement is evaluated only once and if it does not evaluate to `true`, the statement is treated as a skip statement. To ensure that the guards are continuously evaluated and that triggers are picked up, a goto statement is added after the if statement that jumps back to the if statement.

Last, the if statement in NQC requires an expression as condition and not a statement as is the case with Promela. For transitions with guards this is no problem, since these are expressions in SLCO as well. When a transition has no guard or trigger, the transition is always enabled. Therefore, the expression `true` is used in this case. Triggers should also be transformed into expressions. A deadline is transformed into an expression that checks whether a timer has exceeded the deadline value. This timer is started when the transition is taken to the state from which an outgoing transition has a deadline trigger. A signal reception is transformed into an expression that checks whether the expected message is ready to be received. Note that this is possible since an RCX stores the last message that has been received.

An RCX can only send and receive integer values over its infrared port. Since signals in SLCO consist of the signal itself and possibly a number of arguments, they have to be encoded such that they can be represented as a single integer. This means that before a signal is sent, it has to be encoded and that after a signal has been received it has to be decoded. The actual sending and receiving of messages is done via API calls.

An example of the transformation from SLCO to NQC is depicted in Figure 7. Note that this example differs from the examples for POOSL and Promela since depicting the translation of message encoding and decoding would clutter the example.



(a) Partial state machine

```
S0:
 if(x == 0){x = 1;
            goto S1;
           }
 if(x == 1){x = 0;
            goto S2;
           }
 goto S0;
```

(b) Corresponding NQC code

Figure 7: Transforming SLCO to NQC

3.3 Implementation

To enable creation of SLCO models, we implemented a metamodel for SLCO using the Eclipse Modeling Framework (EMF) [24]. This implementation serves as a basis for the model transformations from SLCO to the other formalisms. Unfortunately, creating large models using the standard tree-view editor provided by EMF is cumbersome. To ease the process of modeling, we implemented a prototype of a graphical modeling environment using the Eclipse Graphical Modeling Framework (GMF) [1]. We also defined a textual syntax for SLCO using xText [9], which provides us with a textual SLCO editor. In this way, there are three ways to create SLCO models. The look and feel of each of these model editing facilities is different since the concrete syntax is different. However, the underlying metamodel representing the abstract syntax is the same.

All transformations that are used to bridge the platform gaps are implemented using the Xtend model transformation formalism [12]. In total there are 15 model transformations consisiting of a total of 207 transformation functions. To enable transformations to POOSL, Promela, and NQC, we implemented metamodels for these languages using EMF as well. The transformations from SLCO to POOSL and NQC are also implemented using Xtend. The transformation from SLCO to Promela is implemented using the Atlas Transformation Language (ATL) [14]. The result of each of these transformations is a model. These models cannot be used in POOSL, Promela, and NQC directly, since they require a model in textual form. Therefore, we implemented model-to-text transformations for these tools using Xpand [12].

4. LANGUAGE EVOLUTION

Van Deursen et al. identify three phases in the development of a DSL: the analysis phase, the implementation phase, and the phase in which the DSL is used [31]. Mernik et al. split the analysis phase into an analysis and a design phase [18]. Over time, our DSL and the accompanying transformations have evolved. The evolution of the language and the transformations has been influenced by a number of roles, each of which is responsible for performing certain activities that belong to the four phases in the development of a DSL mentioned above. Table 2 shows in which phases of the development each of the roles participate.

The remainder of this section starts with a description of the roles and the activities that belong to these roles. After these descriptions, the major changes made in both the language and the transformations are listed. At the end of this section, we cluster these changes and distinguish three types of causes for DSL evolution.

Development phases	Roles
Analysis	Language designer Platform expert
Design	Language designer
Implementation	Transformation implementor
Use	Modeler

Table 2: Development phases and the corresponding roles

4.1 Roles

The design of our DSL has been influenced by a number of roles. Although each role has its own separate tasks, these tasks greatly depend on each other, which leads to interaction between the roles. The language designer, the platform experts, the modeler, and the transformation implementor each play a role in the evolution of our DSL. The language designer is responsible for designing the syntax and semantics of the language. SLCO models are transformed to a number of platforms. For each of these platforms, a platform expert is responsible for the mapping from SLCO to the platform. Such an expert investigates whether all constructs of SLCO have a counterpart on the target platform. If such counterparts do not exist and cannot be simulated using other constructs, the platform expert and the language designer will consider whether the source language needs to be adapted. In some cases, not all constructs need to have an equivalent counterpart in the target platform. The language and tools offered by the target platform have a certain purpose and some constructs of SLCO may be irrelevant for the purpose of the target platform. In our case, a POOSL expert, a Spin expert, and an NQC expert play a role. The modeler uses the language designed by the language designer to specify systems. Given the mappings from SLCO to a target platform designed by the platform experts, the transformation implementor is responsible for the translation of the conceptual mapping to an actual implementation of the transformation.

4.2 Activities

Table 3 shows how the aforementioned roles are related to activities. The table also shows the number of persons that played the roles. SLCO has been used in a number of student assignments in which students had to create models. Therefore, the actual number of modelers is higher than two. However, little to no feedback was acquired from the students. Because their influence on the evolution of SLCO is negligible, they are not explicitly mentioned in the table.

Role	Persons	Activities
Language designer	4	Defining the language
Platform expert	4	Defining a mapping from SLCO to a platform Interpreting models
Modeler	2	Creating models
Transformation implementor	2	Implementing the mappings Creating models

Table 3: Roles and the corresponding activities

One activity is the definition of the syntax and semantics of the language. This activity is performed by the language designer.

The second and third activity are concerned with the translation of SLCO models to other platforms. First, a mapping from SLCO constructs to the constructs offered by a target platform must be defined. These mappings are created by the platform experts. In some cases, this calls for changes in the language design, because the constructs offered by the language may not match the requirements of the target platform. The language designer extends the language to match these requirements. Once a mapping from SLCO constructs to platform constructs is defined, a transformation implementing this mapping must be implemented. This is the task of the transformation implementor.

The fourth activity is creating models. Models can be created in two ways. New models can be constructed by hand and can be produced by applying a transformation to an existing model. The modeler creates models manually and the transformation implementor creates them by transforming existing models. Both methods have shown to reveal shortcomings of the language design which will be discussed in 4.3.

The fifth activity is interpreting models. The platform experts have to interpret the models created using the mapping they defined, to see whether the results are as expected. In our approach, a transformation from an SLCO model to a model in one of the target platforms consists of a number of steps. The result of each intermediate step is another SLCO model. These models are also interpreted by the platform expert.

4.3 Evolution of the Language

Table 4 shows how the activities described in Section 4.2 have influenced the language. This section discusses each of these influences.

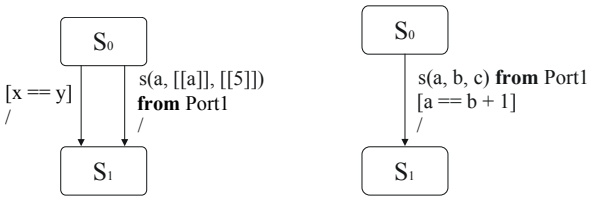
Activity	Language features
Defining the language	Concurrent, communicating objects UML-like syntax Synchronous signals
Defining a mapping from SLCO to a platform	Asynchronous signals Lossy channels Conditional signal reception Triggers and guards
Interpreting models	Conditional signal reception Local variables Additional operators and types Fine-grained transformations
Creating models	Initial values of variables Identifying signals by name
Implementing the mappings	Explicit channel types

Table 4: Desired language features identified during each of the activities

From an abstract point of view, the problem domain for which we designed a DSL consists of concurrent, communicating objects. The most important requirement for SCLO is therefore that it can describe such objects at an appropriate level of abstraction. One purpose of SLCO is using the language for documentation. For this reason, a graphical syntax resembling the well-known syntax of the UML was chosen. Another design decision we made for this reason is offering communication via synchronous signals. This form of communication leads to concise models, which increases the understandability of models and thus increases their value as documentation.

Communication between RCXs occurs using infrared signals. These signals are asynchronous and the channel used for this type of communication is unreliable. Initially, communication in SLCO could only occur using synchronous channels over reliable channels. To enable a straightforward mapping from SLCO to NQC, the characteristics of both languages should be aligned. Therefore, SLCO was extended with asynchronous signals and lossy channels. Additionally, two mappings were defined to achieve this alignment (see Sections 3.1.1 and 3.1.3). Adding these concepts and mappings enables simulation and verification of SLCO models on a level of abstraction that is close to that of the NQC code that is executed.

During the interpretation of Promela models, we noticed that the state space generated during the model checking process could be reduced by adapting the language. Reducing the state space improves verification performance. We adapted SLCO by introducing conditional signal reception as a language construct. Conditional signal reception can be used to specify that a signal can only be received if one or more of its arguments have a fixed value. The rightmost transition in Figure 8(a) , for example, will only be triggered when a signal s is received whose second argument equals the current value of a and whose third argument equals 5. If these conditions hold, the transition is triggered and the value of the first argument of the signal is stored in variable a.



(a) Conditional signal reception **(b) Transition with a trigger and a guard**

Figure 8: Triggers and Guards

While investigating different forms of conditional signal reception and their mapping to the target platforms, we noticed that not all combinations of triggers and guards can be translated to Promela. The combination shown in Figure 8(b) , for instance, has no counterpart in Promela that matches the semantics of SLCO. We dealt with this problem by disallowing a transition to have both a trigger and a guard. In practice, this restriction posed no problems because we have never used a trigger in combination with a guard in our models.

During the interpretation of SLCO models, we noticed that some variables were used only locally by state machines, but were defined globally as variables of the class containing the state machines. To improve the readability of the models, we introduced the concept of local variables. However,

this increase in readability is accompanied by a decrease in modifiability. We use the Eclipse Modeling Framework to implement our DSL and transformations. In this framework, a tree view editor is a commonly used tool for editing models. When referring to a variable in this tree view editor, the container of a variable is not shown in the list of variables that can be referred to. This makes it hard to distinguish between two variables with the same name but with a different container. The prototype of the graphical editor suffers from the same problem. Another improvement in terms of readability and modifiability we made after interpreting SLCO models is adding new operators and types to the modeling language.

While creating models by hand, we noticed that it was tedious to initialize variables explicitly as part of a state machine describing the behavior of a class. For this reason, we made it possible to specify the initial value of a variable as part of its declaration. Another design decision that was tedious to deal with while creating models by hand was the existence of a metaclass for signals. In the current version of SLCO, this metaclass is replaced with a simple attribute of type String that denotes the name of a signal.

In SLCO, the arguments of all signals sent over a channel must have the same type. A channel in SLCO is characterized by these types, its directionality, its synchronicity, and its reliability. Initially, the allowed types of the arguments and the directionality of a channel where left implicit. However, the types of the arguments of the signals sent over a channel must be specified explicitly in Promela. When transforming SLCO to Promela, the characteristics of a channel had to be derived from the statements that use the channel for sending and receiving. To avoid this, we decided to make the type of channels explicit, which lead to smaller transformations that were easier to understand and modify.

4.4 Evolution of the Transformations

The transformations from SLCO to the various platforms have also evolved. One reason for this evolution was the decision to divide each transformation into steps that either modify existing elements of a model, or add elements to a model. We chose this approach, because it clarifies the relation between two consecutive steps of a transformation. It is now easier to see what has changed after applying a step of the transformation.

Another reason for splitting up some of the transformations into smaller steps is to enable verification of intermediate models that are closer to the implementation. This is schematically depicted in Figure 9. Before splitting the

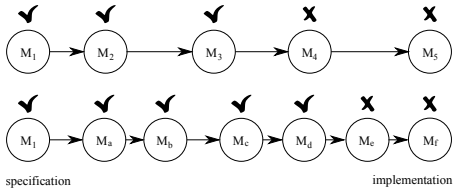


Figure 9: Fine-grained transformations

transformations into smaller parts, models M_1, M_2, and M_3 could be verified using model checking. Verifying model M_4 was already impossible due to the state-space explosion problem. After splitting, verification up to model M_d is possible. This model is much closer to the implementation than model M_3. Splitting the transformation increased the usability of the models.

4.5 Generalization

Looking more closely at the evolution of our language and transformations, we noticed that the design choices and changes can be divided into four categories. We consider these categories to characterize the main influences on the evolution of our DSL and transformations.

The choice of defining a language for the specification of systems consisting of concurrent objects is made to effectively describe problems in the *problem domain*. The influence of the problem domain on the evolution of a DSL is to be expected, since the language is focused on that domain.

In our case, the *target platforms* also have an impact on the evolution, since our language does not have its own execution platform, nor is it embedded in another language. If a transformation to, for example, an execution platform is impossible due to language mismatches, it is impossible to execute a model created using the DSL. Therefore, our DSL has evolved to eliminate unresolvable mismatches with the target platforms. The introduction of conditional signal reception and the division of the transformations into smaller steps are enforced by the target platforms.

A large number of design decisions have been made to increase the *quality of models*, in particular their understandability and modifiability. Defining a graphical syntax for the language, adding synchronous communication, introducing local variables, and allowing variables to have an initial value have been done to increase the understandability. Referring to signals based on their name has increased the modifiability of models. The many changes required for increasing the understandability and modifiability of our DSL can be partially explained by our lack of experience in designing DSLs. We found that it is hard to predict which language features will improve understandability and modifiability without actually using the language. These are, however, important quality attributes that should be taken into account to enhance the chance for acceptance of the language in practice. We expect that there are more quality attributes [5] that influence the evolution of a language, which will become apparent when the language is used more extensively.

Enhancing the *quality of the transformations* also had its effect on the language. To increase the understandability and modifiability of the transformations, certain implicit language features have been made explicit, such as the directionality of channels and the argument types of the signals sent over channels.

5. RELATED WORK

Freeman and Pryce describe their experiences with the evolution of a DSL [11]. Their language has evolved from a library on top of Java to an embedded language. We observed that our DSL has evolved because new target platforms were added. Their language is an embedded language that does not need target platforms. Because the language is embedded, it is highly influenced by the host language, Java in this case. One of the reasons for the evolution of their language is to improve user-friendliness. This is related to our observation that language evolution is influenced by model quality.

The evolution of a visual programming language for writ-

ing real-time control programs for distributed environments and supporting tools is described by Karaila [15]. The main goal of this language is to make programs understandable. The author shows that Lehman's eight laws [16] for characterizing the way software systems tend to evolve are not restricted to software systems, but apply to language evolution as well. The main difference between this language and SLCO is the scale. The language studied by the author has been under development since 1988 and the last major development step dates from 2003. Also the approach of the study is different. Karaila tries to fit his experiences within a framework, whereas we try to distill general lessons from our own experience.

An analysis of over twenty cases where domain-specific languages were designed to implement a model-driven engineering approach is presented by Luoma et al. [17]. The data for the analysis was gathered by means of interviews and discussions with people involved in constructing the DSLs. They identified and categorized four influences on defining DSLs: domain concepts, generated output, look and feel of the system built, and expression of variability. One of the differences with our work is, again, the scope of the study. They studied the definition of over twenty languages, of which some have been used for several years already. We defined only one language, but our focus is on the evolution of the language. Another important difference is that in their case the target platform for the applications developed with the DSL was already chosen, whereas we added multiple target platforms on the fly.

Van Deursen and Klint discuss experiences from industrial practice regarding the development of domain-specific languages [30]. The focus of the article is on the development of a DSL for describing financial products. They provide guidelines and considerations that should be taken into account when developing a DSL. Issues as maintainability factors and risks involved in the use of DSLs are also addressed. The DSL they developed has been used for a few years and during those years it has been concluded that the language needed some changes to increase user friendliness. This is in line with our observation that evolution of a language is influenced by model quality.

A domain-specific visual language that aims at expressing the evolution of domain-specific visual languages is developed by Sprinkle and Karsai [23]. This language and accompanying tools assist in the co-evolution of domain, DSL and models created using the DSL. They claim that what sets the evolution of domain-specific languages apart is that its primary aim is not backward compatibility but domain specificity. This is in line with our observation that the evolution of a domain-specific language is influenced by the evolution of the problem domain. They focus on tools for supporting DSL evolution, rather than researching the causes for language evolution as we do.

Another framework aimed at supporting DSL evolution is described by De Geest [8]. The primary influence on DSL evolution mentioned by the author is use of the DSL. He claims that by using a DSL, more domain knowledge is gathered, which requires adaptation of the DSL. Also, new features may be requested to ease the modeling process. This is in line with our observation that evolution of a language is influenced by model quality.

6. CONCLUSIONS

We identified four main influences on the evolution of our DSL: the problem domain, the target platforms, model quality, and model transformation quality. The problem domain, model quality, and transformation quality continuously influence the evolution of a language throughout the design process. The problem domain should always be taken into consideration when adapting the language to ensure that the abstractions provided by the language fit the domain. Opportunities to adapt the language in order to improve the quality of models and transformations become apparent as experience with the language grows, while designing and also while using the language. Because quality is a subjective concept, quality attributes can be in conflict. In our case, we added local variables to state machines to increase understandability, which had a negative effect on modifiability.

If the purpose of a DSL changes, transformations to platforms that suit this purpose may be required. There may be mismatches between the DSL and the target platform that preclude straightforward transformation. In such cases, the restrictions imposed by the target platform may require the DSL to change. Our expectation was that our DSL would become increasingly more general each time a new target platform was incorporated. In practice, the DSL became more specific. For example, while investigating different forms of conditional signal reception, we had to disallow the simultaneous occurrence of a guard and a trigger on a transition.

7. FUTURE WORK

The main threat to the validity of our research is the scale. The language provides only a limited amount of modeling constructs and we implemented only a limited number of model transformations. Since we find conclusions similar to our own in literature, we expect that our conclusions will also hold for larger scale DSL projects. However, researching the evolution of a more extensive DSL to experience whether different evolution issues arise is relevant future work.

The correctness of the implemented model transformations has not been proven. We use testing to establish whether the result of a model transformation is as we expect. Instead, we would like to prove the correctness of our model transformations. In order to give a correctness proof, the semantics of source and target language must be defined. We can define the formal semantics of SLCO in the way that best suits our needs. This means that we can define the semantics in a way that makes proving the correctness of our model transformations easier. This could also be an influence on the evolution of the DSL.

Acquiring correct models can also be achieved by adding constraints to the language definition. In this way, models can be validated. This kind of validation can also be used to convince ourselves of the (partial) correctness of the model transformations. A model that is valid before transformation, should also be valid afterwards.

Acknowledgement

We would like to thank Robert-Jan Bijl for his work on the graphical editor.

8. REFERENCES

[1] The Eclipse Graphical Modeling Framework (GMF). http://www.eclipse.org/modeling/gmf/.

[2] Lego Mindstorms website. http://www.lego.com/eng/education/mindstorms/.

[3] J. Baeten and C. A. Middelburg. *Process Algebra with Timing*. Springer, 2002.

[4] D. Baum. *NQC Programmer's Guide*, 2003. http://bricxcc.sourceforge.net/nqc/doc/.

[5] B. W. Boehm, J. R. Brown, H. Kaspar, M. Lipow, G. J. Macleod, and M. J. Merrit. *Characteristics of Software Quality*. North-Holland, 1978.

[6] A. Cicchetti, D. Di Ruscio, R. Eramo, and A. Pierantonio. Automating co-evolution in model-driven engineering. In *Proceedings of the 2008 twelfth International IEEE Enterprise Distributed Object Computing Conference*, pages 222–231. IEEE Computer Society, 2008.

[7] E. M. Clarke Jr., O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 1999.

[8] G. de Geest. Building a framework to support domain-specific language evolution. Master's thesis, Delft University of Technology, The Netherlands, 2008.

[9] S. Efftinge and M. Völter. oAW xText: a framework for textual DSLs. In *Workshop on Modeling Symposium at Eclipse Summit*, 2006.

[10] W. H. J. Feijen and A. J. M. van Gasteren. *On a Method of Multiprogramming*. Springer, 1999.

[11] S. Freeman and N. Pryce. Evolving an embedded domain-specific language in java. In *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, pages 855–865. ACM, 2006.

[12] A. Haase, M. Völter, S. Efftinge, and B. Kolb. Introduction to openArchitectureWare 4.1.2. In *Model-Driven Development Tool Implementers Forum (co-located with TOOLS 2007)*, 2007.

[13] G. J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley, 2003.

[14] F. Jouault and I. Kurtev. Transforming models with ATL. In *MoDELS 2005 Satellite Events*, number 3844 in Lecture Notes in Computer Science, pages 128–138. Springer, 2005.

[15] M. Karaila. Evolution of a Domain Specific Language and its engineering environment – Lehman's laws revisited. In *Proceedings of the 9th OOPSLA Workshop on Domain-Specific Modeling*, 2009.

[16] M. M. Lehman, J. F. Ramil, P. Wernick, D. E. Perry, and W. M. Turski. Metrics and laws of software evolution – The nineties view. In *Proceedings of the 4th IEEE International Software Metrics Symposium*, pages 20–33. IEEE Computer Society, 1997.

[17] J. Luoma, S. Kelly, and J.-P. Tolvanen. Defining domain-specific modeling languages: Collected experiences. In *Proceedings of the 4th OOPSLA Workshop on Domain-Specific Modeling*, number TR-33 in Computer Science and Information System Reports. University of Jyväskylä, Finland, 2004.

[18] M. Mernik, J. Heering, and A. M. Sloane. When and how to develop domain-specific languages. *ACM Computing Surveys*, 37(4):316–344, 2005.

[19] R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.

[20] A. Pnueli. The temporal logic of programs. In *Proceedings fo the 18th Annual Symposium on Foundations of Computer Science*, pages 46–57, 1977.

[21] N. Pontisso and D. Chemouil. TOPCASED combining formal methods with model-driven engineering. In *Proceedings of the 21st IEEE International Conference on Automated Software Engineering*, pages 359–360. IEEE Computer Society, 2006.

[22] D. C. Schmidt. Model-driven engineering. *Computer*, 39(2):25–31, 2006.

[23] J. Sprinkle and G. Karsai. A domain-specific visual language for domain model evolution. *Journal of Visual Languages & Computing*, 15(3-4):291–307, 2004.

[24] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks. *EMF: Eclipse Modeling Framework*. Addison-Wesley Professional, second edition, 2009.

[25] B. D. Theelen, O. Florescu, M. C. W. Geilen, J. Huang, P. H. A. van der Putten, and J. P. M. Voeten. Software/hardware engineering with the Parallel Object-Oriented Specification Language. In *Proceedings of the 5th IEEE/ACM International Conference on Formal Methods and Models for Codesign*, pages 139–148. IEEE Computer Society, 2007.

[26] M. F. van Amstel, M. G. J. van den Brand, Z. Protić, and T. Verhoeff. Transforming process algebra models into UML state machines: Bridging a semantic gap? In *Proceedings of the first International Conference on Model Transformation*, volume 5063 of *Lecture Notes in Computer Science*, pages 61–75. Springer, 2008.

[27] D. A. van Beek, P. Collins, D. E. Nadales Agut, J. E. Rooda, and R. R. H. Schiffelers. New concepts in the abstract format of the Compositional Interchange Format. In *Proceedings of the 3rd IFAC Conference on Analysis and Design of Hybrid Systems*, 2009.

[28] D. A. van Beek, A. T. Hofkamp, M. A. Reniers, J. E. Rooda, and R. R. H. Schiffelers. Syntax and formal semantics of Chi 2.0. SE-Report 2008-01, Systems Engineering Group, Department of Mechanical Engineering, Eindhoven University of Technology, The Netherlands, 2008.

[29] M. G. J. van den Brand, L. J. P. Engelen, M. Hamilton, A. Levytskyy, and J. P. M. Voeten. Embedded systems modeling, analysis and synthesis. In *Ideals: evolvability of software-intensive high-tech systems*, chapter 7, pages 99–112. Embedded Systems Institute, Eindhoven, The Netherlands, 2007.

[30] A. van Deursen and P. Klint. Little languages: Little maintenance? *Journal of Software Maintenance*, 10(2):75–92, 1998.

[31] A. van Deursen, P. Klint, and J. Visser. Domain-specific languages: An annotated bibliography. *SIGPLAN Notices*, 35(6):26–36, 2000.

[32] E. Visser. *Syntax Definition for Language Prototyping*. PhD thesis, Universiteit van Amsterdam, The Netherlands, 1997.