# Simple Language of Communicating Objects

Marcel van Amstel, Suzana Andova, Mark van den Brand, and Luc Engelen

May 27, 2013

# Table of Contents

**Table of Contents** **iii**

**1 SLCO** **1**
1.1 Metamodel . . . 1
1.2 Concrete Syntax . . . 3
1.3 Syntactic Sugar for SLCO . . . 7

**2 Target Languages** **9**
2.1 POOSL . . . 9
2.2 NQC . . . 9
2.3 Promela . . . 9

**3 Model Transformations** **11**
3.1 Semantic Gaps and Platform Gaps . . . 11
3.2 Model Transformations . . . 12
3.3 Sequences of Transformations . . . 25

**4 Implementation** **29**
4.1 Implementation . . . 29
4.2 ASF+SDF and the Meta-Environment . . . 30
4.3 openArchitectureWare . . . 32
4.4 ATL Transformation Language . . . 33
4.5 Dot and Graphviz . . . 33

**Bibliography** **35**

**A Operational Semantics of SLCO** **37**
A.1 Introduction . . . 37
A.2 Syntax . . . 37
A.3 Semantics . . . 38
A.4 Initialization . . . 44

# Chapter 1

# Simple Language of Communicating Objects

The Simple Language of Communicating Objects (SLCO) is a small domain-specific modeling language for the specification of systems consisting of objects that operate in parallel and communicate with each other. Via a number of model transformations, SLCO models can be simulated, executed, and verified. In this chapter, we present the language itself, including an informal description of its semantics. A formal semantics of SLCO is presented in Appendix A.

We start by describing a basic version of SLCO in Sections 1.1 and 1.2, to simplify the description of the semantics of the language. Finally, in Section 1.3, the complete version of the language is described. This version makes the task of creating models easier and allows more concise definitions of model transformations. It is a straightforward extension of the basic version of the language.

Over time, SLCO and the related model transformations have evolved. This evolution was first described by Van Amstel et al. [1] and more recently by Engelen [16]. For simplicity, we describe only the most recent version of the language and the related transformations below.

## 1.1 Metamodel

An SLCO model consists of a number of classes, objects, and channels, as shown by the partial metamodel in Figure 1.1. Objects are instances of classes. A class describes the structure and behavior of its instances. It has ports and variables that define the structure of its instances and state machines that describe their behavior. It is possible to specify the initial values of variables. If no initial value is specified, integer variables are initialized to 0, Boolean variables are initialized to **true**, and string variables are initialized to the empty string. The variables of a class are global variables in the sense that they can be used by all state machines that are part of the class. Ports are used to connect channels to objects, and each port is connected to at most one channel. The state machines that are part of a class can only send and receive signals via the ports of this class. In case a class specifies that its instances consist of multiple state machines, these state machines operate in parallel.

A state machine consists of variables, states, and transitions. In contrast to the variables of a class, the variables of a state machine are local variables because they can only be used by the state machine that contains them. SLCO offers two special types of states: initial states and final states. A state machine starts in its initial state, and an SLCO model has successfully terminated when all its constituting state machines have reached a final state. Each state machine has exactly one initial state and can contain any number of ordinary and final states. A transition has a

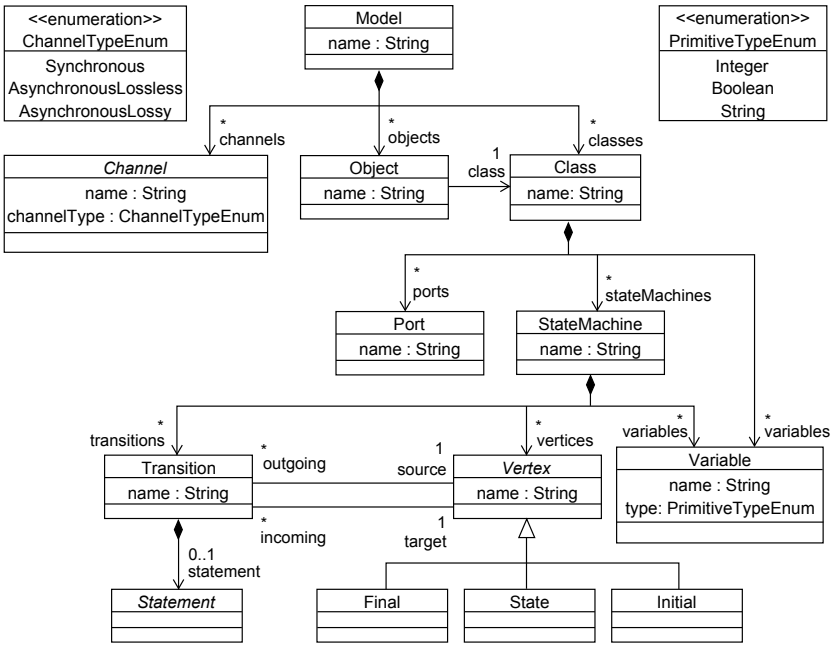


Figure 1.1: Part of the SLCO metamodel containing the main constructs of the language

source and a target state, and can be associated with a single statement. Each statement is either blocked or enabled, and a transition is enabled if it is not associated with any statement or if its statement is enabled. Otherwise, the transition is blocked. Taking an enabled transition from its source state to its target state leads to the execution of the associated statement. A transition that is blocked cannot be taken. The execution of a single statement is atomic. If a transition has multiple outgoing transitions that are enabled, one of them is taken non-deterministically.

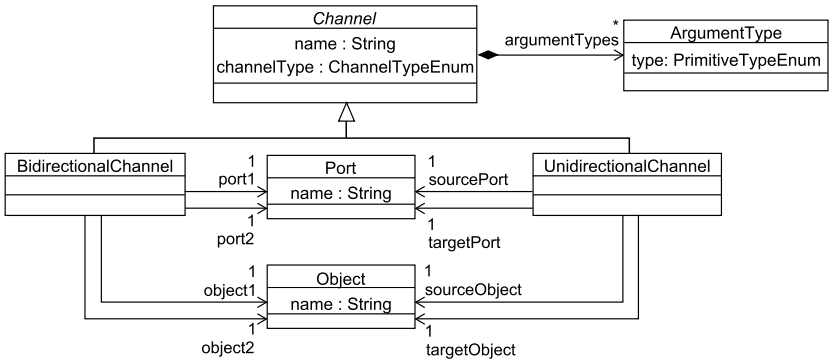


Figure 1.2: Part of the SLCO metamodel containing the constructs related to channels

Objects communicate with each other via channels, which are either bidirectional or unidirectional. SLCO offers three types of channels: synchronous channels, asynchronous, lossy channels, and asynchronous, lossless channels. Each asynchronous channel is implicitly associated to one or two one-place buffers. A unidirectional channel is associated to one buffer, and a bidirectional channel is associated to two buffers, one for each direction. Signals can be sent over an asyn-

chronous, lossless channel in a certain direction if the buffer associated to that direction is empty. If a buffer is not empty, statements that send signals over this buffer block. Signals can always be sent over asynchronous, lossy channels. Because these channels are lossy, however, some signals sent over these channel are not stored in the corresponding buffer. If the buffer corresponding to a channel already contains a signal and another signal is sent over this channel, the existing signal is replaced with the new signal. If the buffer associated to a channel is empty, the signal reception statements that receive signals via this channel are blocked. If the buffer associated to a channel contains a signal and a matching signal reception statement is executed, the signal is removed from the buffer and received by the state machine executing the signal reception statement. A channel can only be used to send and receive signals with a certain signature, which defines the number of arguments of a signal and the types of these arguments. The part of the SLCO metamodel concerning channels is shown in Figure 1.2.

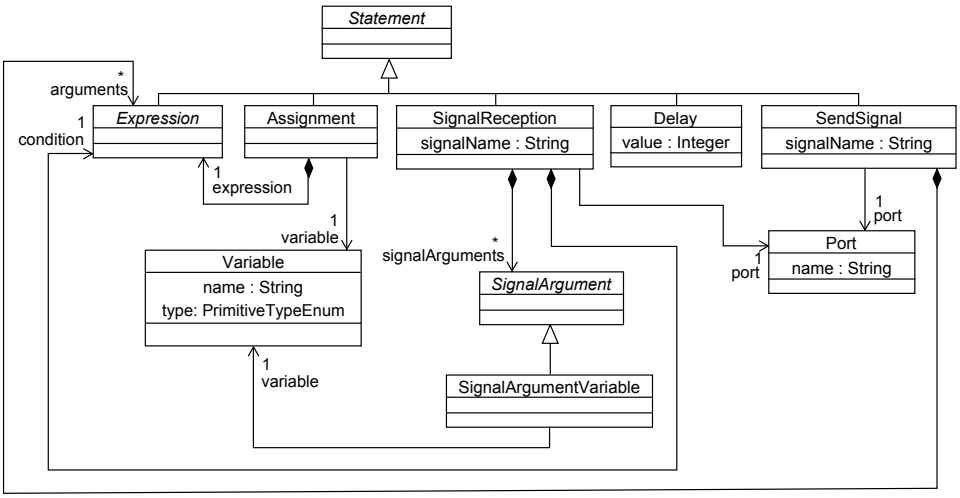


Figure 1.3: Part of the SLCO metamodel containing the constructs related to statements

SLCO offers five types of statements, as shown in Figure 1.3. A Boolean expression represents a statement that blocks the transition from a source to a target state until the expression evaluates to **true**. The part of the SLCO metamodel concerning expressions is shown in Figure 1.4. A delay statement blocks a transition until a specified amount of time measured in milliseconds has passed. A transition with a conditional signal reception statement is enabled if a signal with appropriate arguments is received via the indicated port and the condition associated to the signal reception holds. This condition is a Boolean expression that may refer to the arguments of the signal that is offered via the port. Besides these statements, SLCO also offers statements for assigning values to variables and for sending signals via ports.

## 1.2 Concrete Syntax

SLCO has both a textual and a graphical concrete syntax. The graphical concrete syntax consists of communication diagrams, structure diagrams, and behavior diagrams. The diagrams in Figures 1.5, 1.6, and 1.7 show an example of an SLCO model using the graphical syntax. Below, we describe the model shown in these figures.

The communication diagram in Figure 1.5 shows two objects $p$ and $q$ that communicate over an asynchronous, lossless channel *c1*, an asynchronous, lossy channel *c2*, and a synchronous

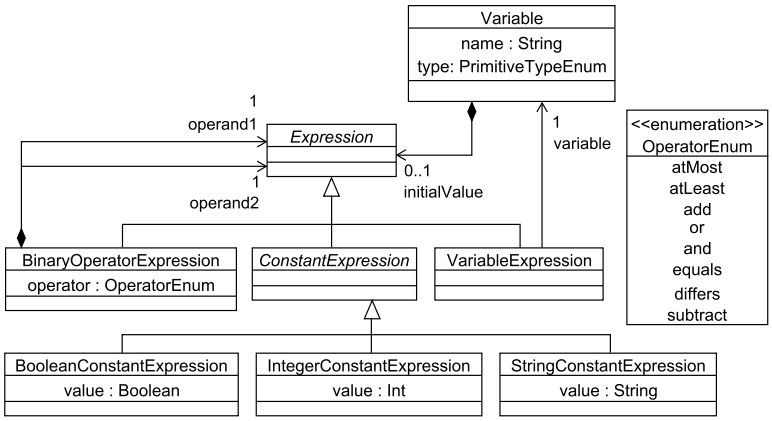


Figure 1.4: Part of the SLCO metamodel containing the constructs related to expressions

channel *c3*. Asynchronous, lossless channels, such as *c1*, are denoted by dashed lines, asynchronous, lossy channels, such as *c2*, are denoted by dotted lines, and synchronous channels, such as *c3*, are denoted by solid lines. The diagram shows that both objects have three ports. The ports of object $p$ are named *In1*, *In2*, and *InOut*, and the ports of object $q$ are named *Out1*, *Out2*, and *InOut*. Arrowheads indicate the directionality of channels: unidirectional channels have one arrowhead and bidirectional channels have two. Object $q$ can only send signals over channel *c1* via its port *Out1*, for example, and object $p$ can only receive signals sent over channel *c1* via its port *In1*. Channel *c3*, however, can be used for communication in both directions. Channel *c1* can only be used to send and receive signals with a Boolean argument, channel *c2* is restricted to signals with integer arguments, and channel *c3* is restricted to signals with string arguments. The figure also shows that object $p$ is an instance of class $P$ and that object $q$ is an instance of class $Q$.

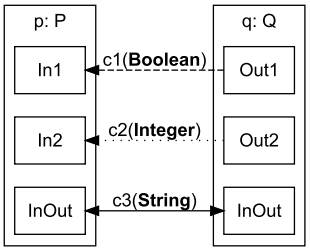


Figure 1.5: Objects, ports, and channels of an SLCO model

The structure diagram in Figure 1.6 shows the structure of classes $P$ and $Q$. Class $P$ consists of three state machines (*Rec1*, *Rec2*, and *SendRec*), and class $Q$ consists of one state machine (*Com*). Furthermore, it shows that class $P$ comprises an integer variable $m$, state machine *Rec1* comprises a Boolean variable $v$, and state machines *SendRec* and *Com* comprise a string variable $s$. The initial value of variable $m$ is 0. The initial values of the other variables are not specified, which means that Boolean variable $v$ is initialized to **true**, and the string variables $s$ are initialized to empty strings, as described above.

The behavior diagram in Figure 1.7 shows the four state machines of our example model. For convenience, the names of the states reflect the names of the state machines that contain them. The three state machines on the left of this figure specify the behavior of object $p$, and the state machine on the right specifies the behavior of object $q$. Initially, all of these state machines are

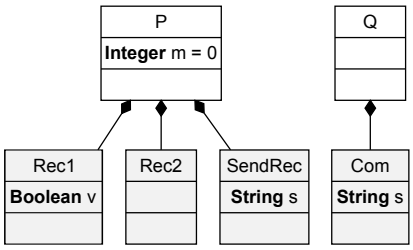


Figure 1.6: Classes, state machines, and variables of an SLCO model

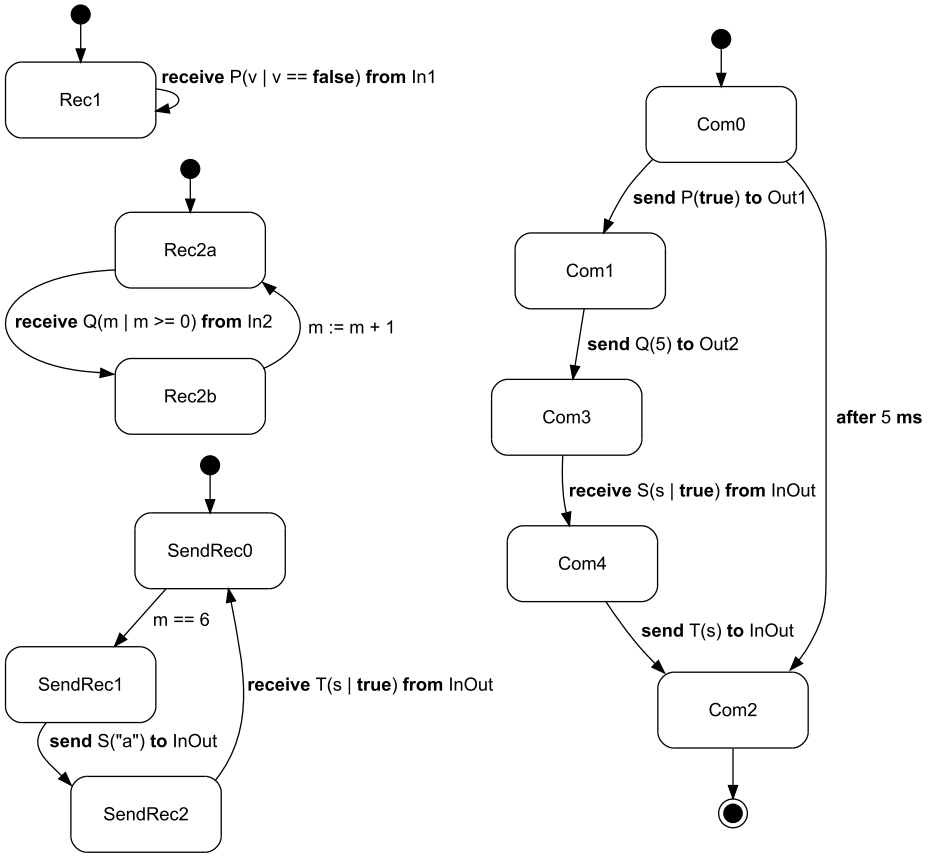


Figure 1.7: State machines of an SLCO model

in their initial states, which are named *Rec1*, *Rec2a*, *SendRec0*, and *Com0*. The fact that these states are initial states is indicated by an incoming arrow from a solid black dot. The black dot itself does not represent a separate state.

When state machine *Com* makes the transition from state *Com0* to state *Com1*, it executes the statement **send** $P$(**true**) **to** *Out1*. The signal sent by object $q$ is never received by object $p$, however, because the statement **receive** $P(v \mid v == \textbf{false})$ **from** *In1* of state machine *Rec1* can only receive signals if their argument equals **false**. After state machine *Com* has been in state *Com0* for 5 ms, the transition from state *Com0* to final state *Com2* is enabled. This means that after 5 ms have passed while state machine *Com* is in state *Com0*, it can non-deterministically

choose to send a signal or terminate. The fact that state *Com2* is a final state is indicated by an outgoing arrow to a circled black dot. Also in this case, the circled black dot itself does not represent a separate state. While making the transition from state *Com1* to state *Com3*, a signal $Q(5)$ is sent to port *Out2* first. Then, after a signal $S(\mathrm{s})$ is received from port *InOut*, state *Com4* is reached. Afterwards, while making the transition from state *Com4* to state *Com2*, a signal $T(\mathrm{s})$ is sent to the same port. Both ports named *InOut* are connected to a synchronous channel, which means that communication via these ports is synchronous. A statement that sends signals via these ports can only be executed when the object on the other side of the channel is able to accept the signal that is being sent. If there is no matching signal reception for a given statement that sends a signal, the latter statement blocks.

The conditional signal reception statement **receive** $Q(m \mid m >= 0)$ **from** *In2* specifies that state machine *Rec2* will only accept a signal $Q(\mathrm{m})$ if the condition $m >= 0$ holds. After receiving such a signal, the value of global variable $m$ is incremented.

The statement $m == 6$ of state machine *SendRec* blocks until the value of global variable $m$ equals 6. Once this condition holds, a signal is sent via port *InOut*, after which the state machine waits for the reception of a signal via the same port.

Listing 1.1: Part of a textual SLCO model

```
1  model CoreWithTime {
2    classes
3      Q {
4        ports
5          Out1 Out2 InOut
6
7        state machines
8          Com {
9            variables
10             String s
11
12           initial Com0
13
14           state Com1 Com3 Com4
15
16           final Com2
17
18           transitions
19             InitialToState from Com0 to Com1 {
20               send P(true) to Out1
21             }
22             ...
23         }
24     }
25     ...
26   objects
27     p: P
28     q: Q
29
30   channels
31     c1(Boolean) async lossless from q.Out1 to p.In1
32     c2(Integer) async lossy from q.Out2 to p.In2
33     c3(String) sync between p.InOut and q.InOut
34 }
```

As mentioned above, SLCO also has a textual concrete syntax. Listing 1.1 shows a part of the example model of Figures 1.5, 1.6, and 1.7 using the textual syntax. The listing shows how each

of the language constructs is represented textually, except for some of the statements. The textual syntax of these statements, however, is exactly the same as in the graphical syntax.

## 1.3 Syntactic Sugar for SLCO

The basic version of SLCO discussed in Sections 1.1 and 1.2 allows a transition to have at most one statement and offers a signal reception statement that is always associated to a condition. In Figure 1.8, a semantically equivalent but more concise version of the behavior diagram of Figure 1.7 is shown, which illustrates the extended version of SLCO. In this figure, some of the transitions are associated to a list of statements, instead of only a single statement. Furthermore, the conditions of signal reception statements are not shown in the figure if they are equal to **true**, and an additional type of conditional signal reception of the form **receive** $P([[\textbf{false}]])$ **from** *In1* is introduced. Because of its conciseness, this version of SLCO is used when creating models and to describe the model transformations in Chapter 3. All of the extensions mentioned above can be expressed in terms of the concepts offered by the basic version of SLCO. Therefore, we call these extensions syntactic sugar for SLCO.

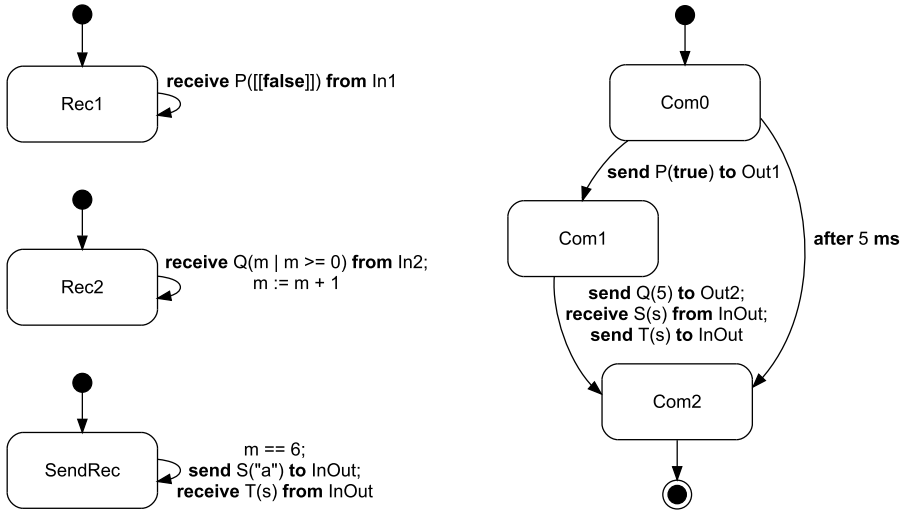


Figure 1.8: State machines of the SLCO model of Figure 1.7 after applying syntactic sugar

The metamodel of the basic version of SLCO has to be adapted to accommodate the changes to the language. These changes are shown in Figures 1.9 and 1.10. In Figure 1.9, the fact that a transition can be associated to a list of statements is shown. When a transition is made from one state to another, the statements that are part of this transition are executed one by one. After executing one of the statements related to a transition, an implicit intermediate state is reached. The execution of statements related to a given transition can be interleaved by the execution of statements related to a transition of a concurrent state machine.

Figure 1.10 shows that the condition of a signal reception statement has become optional. A signal reception statement without condition is equivalent to a conditional signal reception statement with a condition equal to **true**. The figure also shows that an additional type of signal argument has been added to the language. Now, there are two ways of specifying that a signal reception is conditional. First, expressions given as arguments of a signal reception specify that only signals whose argument values are equal to the corresponding expressions are accepted. Second, only those signals are accepted for which the optional condition of a signal reception

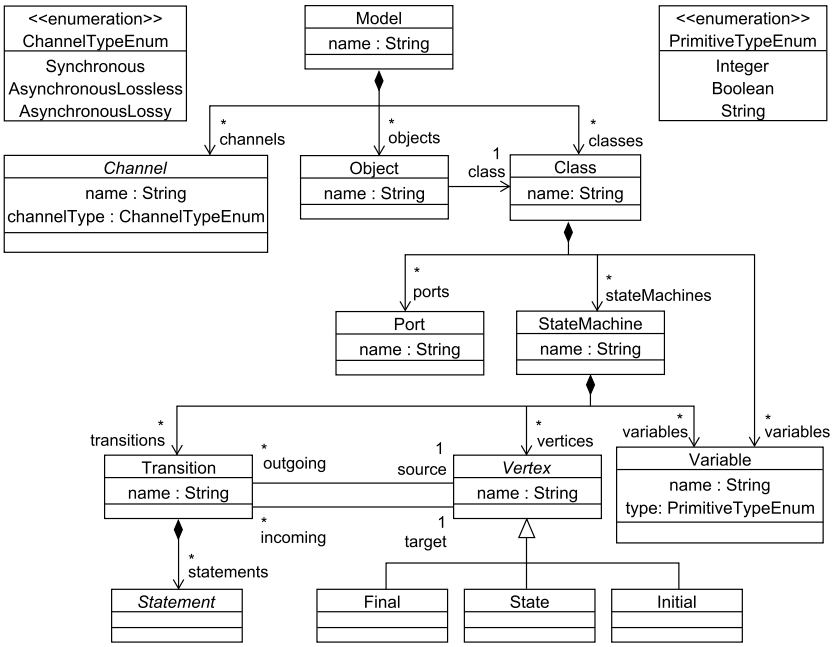


Figure 1.9: Extended version of the part of the SLCO metamodel shown in Figure 1.1

evaluates to **true**. The statement **receive** $P([[\textbf{false}]])$ **from** *In1* illustrates the first type of conditional signal reception. The statement can only receive signals if their argument equals **false**. The double square brackets in the signal reception statement are used to distinguish expressions from variables. For instance, **receive** $S(v)$ **from** $P$ represents a statement that receives signals named $S$ from port $P$ and stores the value of the received argument in variable $v$. In contrast, the statement **receive** $S([[v]])$ **from** $P$ only receives those signals named $S$ from port $P$ whose argument is equal to the value of variable $v$.

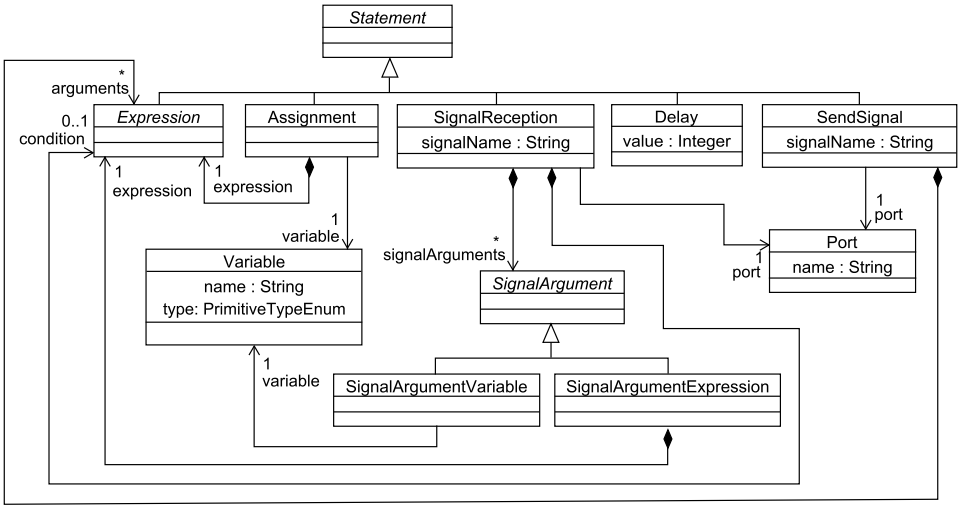


Figure 1.10: Extended version of the part of the SLCO metamodel shown in Figure 1.3

# Chapter 2

# Target Languages

SLCO models can be simulated, executed, and verified by transforming them to equivalent models and implementations in a number of languages. Before describing the transformations from SLCO to these languages in Chapter 3, we discuss the target languages themselves.

## 2.1 POOSL

We use the Parallel Object-Oriented Specification Language (POOSL) [29], 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 [23], and the part that is used 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. SHESim offers interactive simulation of POOSL models using its built-in POOSL interpreter. Rotalumis is a command-line tool that can simulate POOSL models at high speed by compiling them to byte code that can be executed on a virtual machine.

## 2.2 NQC

To execute SLCO models, an implementation platform is required. We chose to use the Lego Mindstorms[1] platform for this purpose. The key part of this platform is a programmable controller 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 controller, instead of the newer and more advanced NXT controller, to investigate the strength of our transformational approach when dealing with a primitive execution platform. The language we use to program these programmable controllers is called Not Quite C (NQC) [8]. 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 via the infrared ports.

## 2.3 Promela

Model checking is an automated verification technique that checks whether a formally specified property holds for a model of a system [12]. We use the model checker Spin [19] for verifying our

[1] `http://mindstorms.lego.com/`

models. Spin can, among others, check a model for deadlocks, unreachable code, and determine whether it satisfies a Linear Temporal Logic (LTL) property [27]. 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. Promela has constructs for modeling selections and loops, based on Dijkstra's guarded commands, and 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 Promela is similar to that of C.

# Chapter 3

# Model Transformations

In this chapter, we describe the languages used for simulation, execution, and verification, and the model transformations related to the language. Additionally, we describe the implementation of the language and its transformations.

## 3.1 Semantic Gaps and Platform Gaps

There are a number of differences between SLCO, POOSL, NQC, and Promela in terms of the language constructs they offer. These differences are often referred to as semantic gaps [3]. Furthermore, the Lego Mindstorms execution platform has practical limitations that poses restrictions to implementations in NQC. These restrictions form platform gaps because they do not hold for models specified using the other languages. We identified a number of gaps between SLCO and the target languages and platforms, which are shown in Tables 3.1 and 3.2. To successfully transform an SLCO model into an equivalent model or implementation in one of the other languages, these gaps have to be bridged.

Table 3.1: Language and platform characteristics related to communication

| Language | (A)synchronous communication | Reliability of communication | Connectivity for communication |
|---|---|---|---|
| NQC | asynchronous | unreliable | broadcast |
| POOSL | synchronous | reliable | point-to-point |
| Promela | both | reliable | point-to-point |
| SLCO | both | both | point-to-point |

Each row in Table 3.1 lists the characteristics related to communication of one of the four languages and the corresponding platform. The second column indicates whether communication is synchronous or asynchronous. In case communication is synchronous, both the sender and receiver of a signal 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 indicates whether communication over channels is reliable. In case a channel is reliable, which is also referred to as lossless, a signal that is sent will always arrive at the receiving end. In case a channel is unreliable, which is also referred to as lossy, a signal that is sent may

get lost. The fourth column shows whether signals are broadcasted or sent using point-to-point communication. When signals are broadcasted, each signal can be received by multiple objects. In the case of point-to-point communication, however, signals are sent from one object to exactly one other object.

Table 3.2: Additional language and platform characteristics

| Language | Support for string constants | Number of Objects |
|---|---|---|
| NQC | no | limited |
| POOSL | yes | $\infty$ |
| Promela | symbolic names for integer constants | $\infty$ |
| SLCO | yes | $\infty$ |

Table 3.2 lists additional characteristics of the four languages and the corresponding platforms. The second column indicates whether a language supports string constants. Although Promela has no support for strings, it offers an enumerated type that allows representing numeric constants using symbolic names, which can be regarded as a restricted form of string constants. The third column lists the amount of objects that can be instantiated simultaneously. In POOSL, Promela, and SLCO, 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 controllers.

## 3.2 Model Transformations

Two types of model transformations are applied to transform SLCO models to models or implementations in one of the target languages: endogenous transformations and exogenous transformations. The input and output language of an endogenous model transformation is the same, whereas exogenous model transformations transform models from one language to another language [22]. The endogenous transformations are used to refine SLCO models by bridging the semantic and platform gaps. After all gaps have been bridged, the straightforward exogenous transformations can be applied.

### 3.2.1 Endogenous Transformations

For each of the semantic gaps and platform gaps described in Section 3.1, we implemented an endogenous model transformation that bridges this gap. To keep these transformations simple, they can only be applied to models adhering to certain conditions. In addition to the transformations that bridge the gaps, we implemented transformations that refine models to ensure that they adhere to the aforementioned conditions. Experiments have shown that developing small transformations has a number of advantages [2, 5], which explains the large amount of endogenous model transformations discussed below.

#### Synchronized Communication over Asynchronous Channels

To bridge the semantic gap between languages offering synchronous communication and languages that do not, we implemented two transformations. Both transformations take an SLCO model

and a synchronous channel as input, and produce a model in which this channel is replaced by an asynchronous, lossless channel and the objects that communicate over this channel are adapted such that they communicate asynchronously. More detailed descriptions of these transformations are given in the PhD thesis of Luc Engelen [16] and in a paper by Andova et al. that discusses the correctness of these transformations [5].

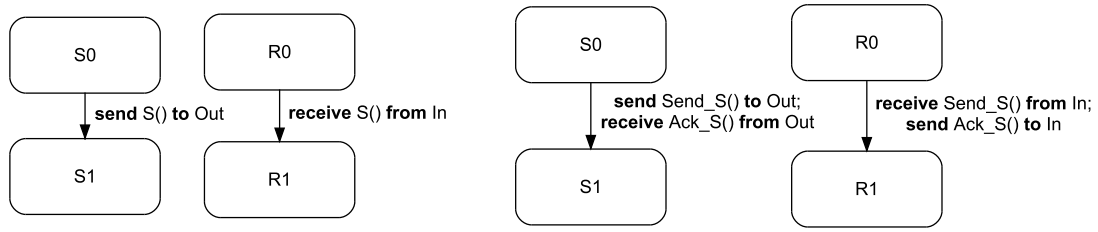


Figure 3.1: Two communicating state machines before and after applying the simple version of transformation $T_{as}$

The first transformation is simple, but can only be applied to models that do not contain states with multiple outgoing transitions if one of these transitions starts with a statement that sends a signal over the synchronous channel. The transformation ensures that the behavior of the model is still as desired by adding acknowledgment signals 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 it receives this acknowledgement. In this way, synchronization is achieved. On the left of Figure 3.1, two partial state machines are shown that send and receive a signal $S$. Initially, ports *In* and *Out* are connected by a synchronous channel. After transformation, acknowledgements are added, as shown on the right of the figure, and the synchronous channel is replaced with an asynchronous, lossless channel.

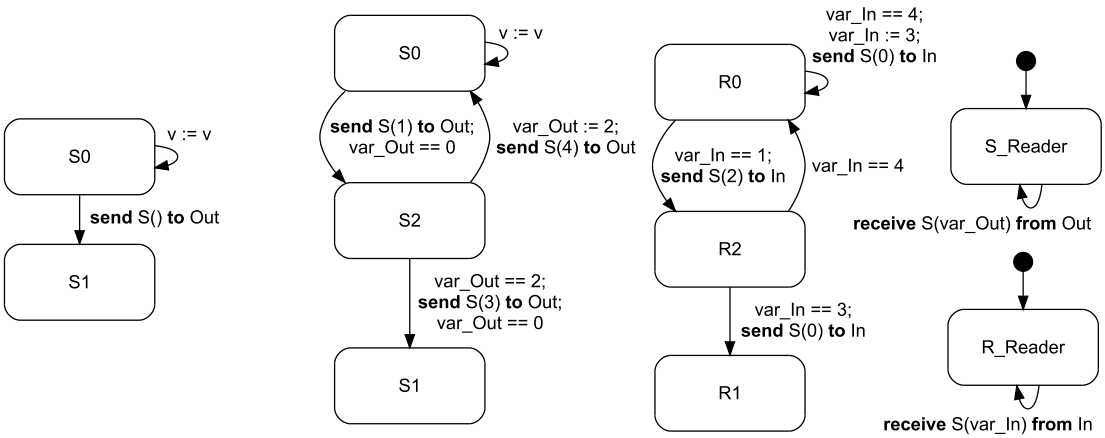


Figure 3.2: Two communicating state machines before and after applying the general version of transformation $T_{as}$

The second transformation can be applied to any SLCO model, but is more complex. The partial state machine on the left of Figure 3.2 shows an example of a situation where the transformation described above cannot be applied. State *S0* has multiple outgoing transitions, and one of these transitions starts with a statement that sends a signal. In such situations, a more complex protocol has to be applied to ensure that the behavior of the model before and after transformation is

the same, apart from the communication over the channel that is replaced. Also in this example, ports *In* and *Out* are initially connected by a synchronous channel, which is replaced by an asynchronous, lossless channel. In Figure 3.2, the second partial state machine from the left shows how the sending state machine is affected by the transformation, and the third partial state machine from the left shows how the receiving state machine is affected. Additionally, two other state machines are added to the model, which are shown on the right of Figure 3.2. State machine *S_Reader* is added to the object that sends signals, and state machine *R_Reader* is added to the object that receives signals. Together with these state machines, the sender and receiver implement a protocol that ensures that states *S1* and *R1* are only reached if the communication between the sender and receiver was successful. Furthermore, as long as the sender and receiver are unable to communicate, all enabled outgoing transitions of states *S0* and *R0* can be made.

The protocol that is employed by this transformation is not straightforward. Therefore, we used a state-space generator for SLCO [4, 16] for feedback during its development. Informally, the protocol consists of the following steps. Initially, the value of variable *var_Out* is equal to 0, and the value of variable *var_In* is equal to 3. First, the sending object sends a signal $S(1)$ to indicate that it wants to communicate. It can proceed to state *S2* if all previous signals have been acknowledged by the receiving object, which is the case if variable *var_Out* is equal to 0. The signal sent by the sending object is received by state machine *R_Reader*, and the value of its argument is stored in *var_In*. Once the receiving object is informed of the intent of the sending object by means of the execution of statement $var\_In == 1$, it sends a signal $S(2)$ to indicate that it is ready to communicate. Once this signal has been received and the value of its argument has been stored in variable *var_Out* by state machine *S_Reader*, the sending object may choose to complete the communication by sending signal $S(3)$. Alternatively, it may choose to cancel the communication by sending signal $S(4)$. Upon receiving one of these signals, the receiving object can take the transition to state *R1* if the communication has been completed successfully, or take the transition to state *R0* if it has been canceled. Either way, it acknowledges the reception of the signal of the sending object by sending a signal $S(0)$. The state machines *S_Reader* and *R_Reader* ensure that the statements that send signals cannot be blocked, by emptying the buffers associated to the channels continuously. It is possible that the sending object sends signal $S(4)$ after sending signal $S(1)$, while the receiving object remains in state *R0*. The self-loop on state *R0* ensures that an acknowledgement is also sent in this situation.

In the remainder, the simple version of this transformation is referred to as $T^S_{as}$, and the general version is referred to as $T^G_{as}$. In cases where it is not relevant which of these two transformations is applied, the abbreviation $T_{as}$ is used.

#### Lossless Communication over a Lossy Channel

Transformation $T_{ll}$ implements lossless communication over a lossy channel by introducing auxiliary objects that implement a concurrent version of the Alternating Bit Protocol (ABP) [7] known as the Concurrent Alternating Bit Protocol (CABP) [6]. This transformation is only applicable to unidirectional channels that are used to communicate signals named *Signal* and whose only argument is a string.

Figure 3.3 shows a model consisting of two objects ($a$ and $b$) that communicate over an asynchronous, lossless channel ($C$). After transformation, channel $C$ is replaced by the channels *c1* to *c6* and four objects that implement the CABP, as shown in Figure 3.4. Object $a$ is connected to an object named *sender*, which communicates over an asynchronous, lossy channel *c2* with an object named *receiver*. The object named *receiver* is in turn connected to the object $b$. After transformation, objects $a$ and $b$ communicate with each other via the aforementioned objects, instead of directly. Objects $a$ and $b$ are connected to these objects by synchronous channels. After

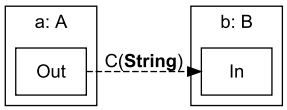


Figure 3.3: Communication diagram showing two objects that communicate over an asynchronous, lossless channel

receiving a signal from object *a*, object *sender* repeatedly sends this signal over channel *c2* until it receives an acknowledgement from object *ar*, to which it is connected via the synchronous channel *c6*. After receiving a signal over channel *c2*, object *receiver* forwards this signal to object *b* and instructs object *as* to continuously acknowledge the reception of this signal. Object *as* does this by continuously sending signals over the asynchronous, lossy channel *c5*. The acknowledgement sent by object *as* contains a two-valued argument that is used by object *ar* to assess whether a particular acknowledgement signal was already received before. Once object *ar* has received a new acknowledgement, it notifies object *sender*. After receiving such a notification, object *sender* is able to receive a new signal from object *a* and transmit this signal over channel *c2*.

In Figure 3.5, the four state machines are shown that specify the behavior of the four objects that implement the CABP. To show which state machine is part of which class, the names of the states have been chosen such that they reflect the names of the classes and objects.

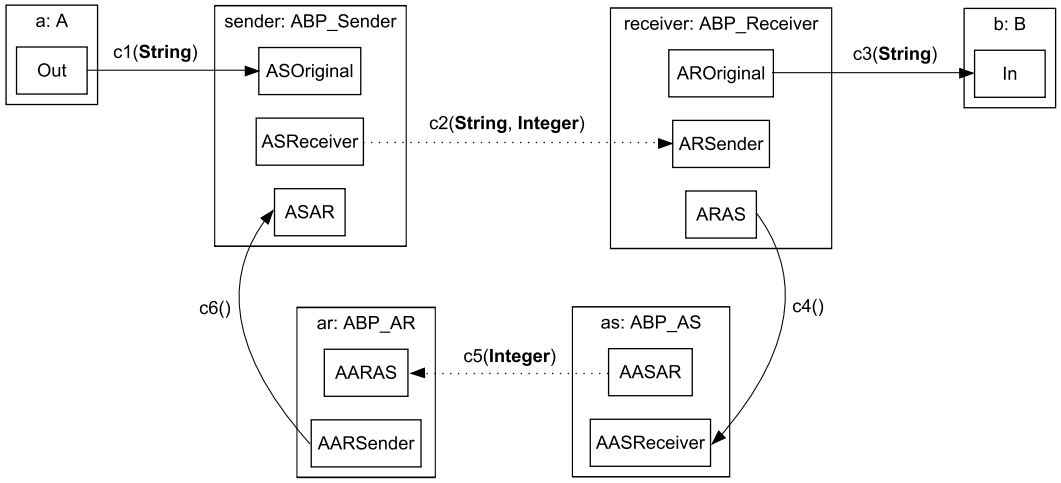


Figure 3.4: Communication diagram showing two objects *a* and *b* that communicate via the CABP

### Adding Delays to Transitions

Transformation $T_{time}$ takes a model and a set of transitions as input, and adds delay statements to these transitions. This transformations is used to control the frequency of the acknowledgments sent by the objects implementing the CABP. Because it reduces the number of signals that are sent, it also reduces the number of collisions between messages sent via infrared on the Lego Mindstorms platform.

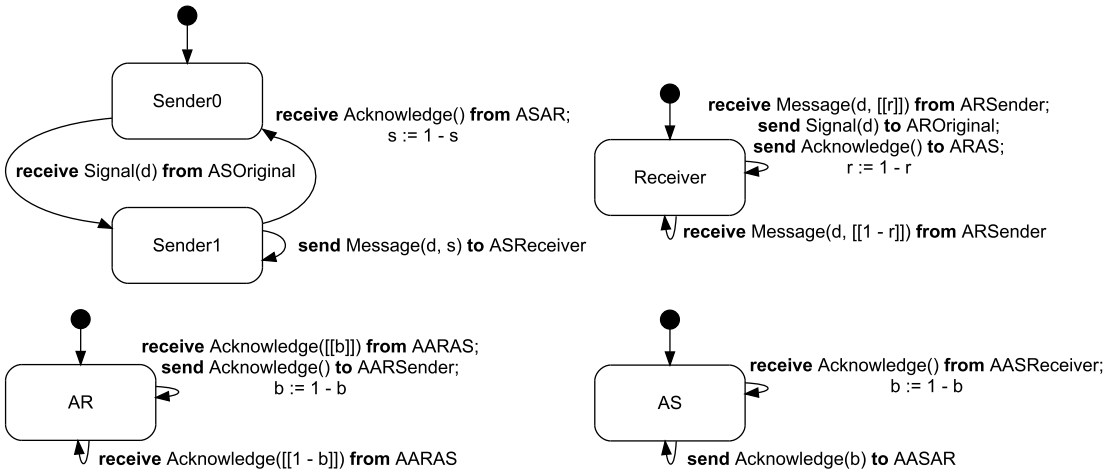


Figure 3.5: Four state machines implementing the CABP

### Replacing Strings by Integers

Transformation $T_{int}$ replaces each string constant in an SLCO model with a unique integer constant and changes the type of all string variables and arguments to integer. This transformation deals with the fact that NQC does not offer strings.

### Making the Sender of a Signal Explicit

When multiple objects broadcast signals with the same name and number of arguments over the same medium, the receiving object cannot determine the origin of such a signal. This situation arises when multiple RCX controllers communicate with each other, because they communicate by broadcasting messages via infrared. To enable a receiving controller to determine the origin of each signal it receives, transformation $T_{ic}$ can be applied to a model. This transformation takes a model and a set of channels as input, and adds an index to all signal names that identifies the channel over which these signals are sent.

### Reducing the Number of Objects

Transformation $T_{merge}$ merges multiple objects into one object. Given a model and a set of objects, it creates a new object that contains all the variables, ports, and state machines contained by the objects provided as input. By reducing the number of objects in a model, it bridges the corresponding gap between SLCO and NQC. If any of the objects that are being merged communicate over synchronous channels, then this form of communication is replaced by communication using shared variables. Transformation $T_{merge}$ is only applicable to objects that satisfy the following condition: each pair of state machines that are part of two communicating objects must communicate over a unique unidirectional, synchronous channel.

Figure 3.6 shows how communication over a synchronous channel is replaced by communication using shared variables. The two partial state machines on the left of the figure are part of two separate objects and communicate with each other by sending and receiving signals over a synchronous channel that connects ports *In* and *Out*. After merging these two objects, the state machines are adapted as shown on the right of the figure and communicate using the shared variables $C\_name$ and $C\_abl$. Variable $C\_name$ is used to store and retrieve the names of

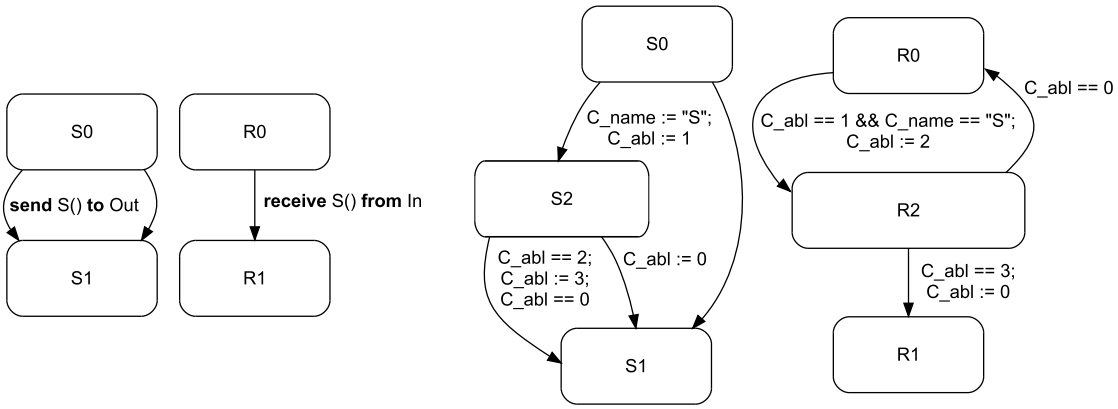


Figure 3.6: Two communicating state machines before and after merging objects

the signals that are being exchanged, and variable $C\_abl$ encodes the states of the employed communication protocol. The sending state machine sets the value of $C\_abl$ to 1 to indicate that it wants to communicate. The receiving state machine indicates that it is also able to communicate by setting the value of $C\_abl$ to 2. If both state machines are able to communicate, the sending state machine can complete the communication process by setting $C\_abl$ to 3. It may also choose to cancel the communication by setting $C\_abl$ to 0. The receiving state machine acknowledges successful completion of the communication process by setting $C\_abl$ to 0.

### Making all Signal Names Equal

To keep the transformation that adds the CABP as simple as possible, our implementation of the CABP takes signals with a fixed name as input, transfers them over a lossy channel, and delivers them at the receiving end. Before this instance of the CABP can be used to substitute an asynchronous, lossless, unidirectional channel, the signal names that are sent over this channel have to be changed into this fixed name. Transformation $T_{arg}$ adapts signals such that their name is changed into this fixed name and the name of the original signal is sent as an argument of the resulting signal. For example, the statement **send** $Block()$ **to** $O$ is replaced by the statement **send** $Signal(\text{“}Block\text{”})$ **to** $O$.

### Replacing a Bidirectional Channel by two Unidirectional Channels

Our implementation of the CABP can only substitute asynchronous, lossless, unidirectional channels. In some cases, therefore, a transformation is needed that replaces communication over a bidirectional channel by communication over two unidirectional channels before transformation $T_{ll}$ can be applied. Transformation $T_{uni}$ performs this task.

### Exclusive Channels for Pairs of State Machines

Transformation $T_{merge}$ cannot merge objects if multiple state machines that are part of an object communicate via the same port. To modify models that do not adhere to this condition, we implemented a transformation $T_{ex}$ that replaces a channel between a pair of objects with a number of identical channels. For each pair of communicating state machines that are part of the two objects, a channel is introduced.

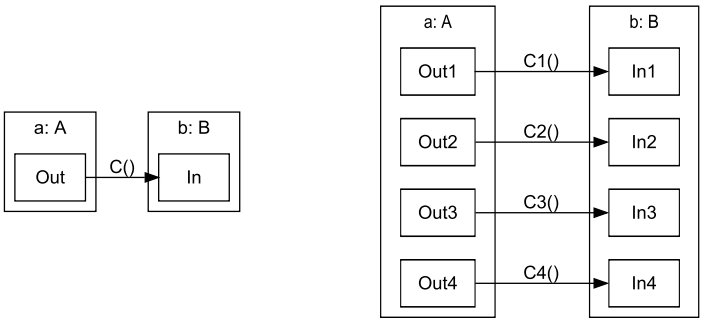


Figure 3.7: Communication diagrams of a model before and after adding exclusive channels

On the left of Figure 3.7, two objects are shown that communicate over a single channel. Both objects contain two state machines (which are not shown in this type of diagram) that communicate over this channel. After applying transformation $T_{ex}$, the channel is replaced by four channels, as shown on the right of Figure 3.7.

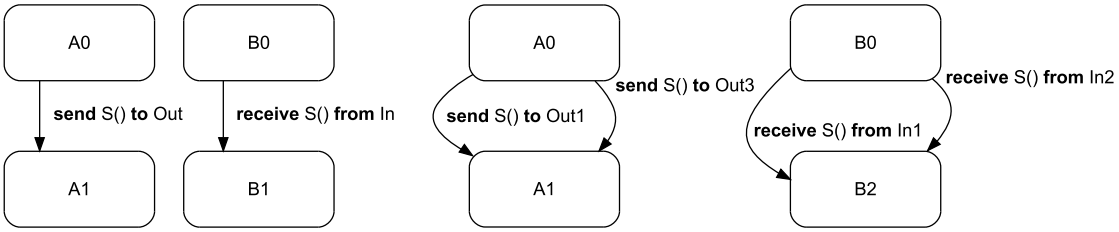


Figure 3.8: Two communicating state machines before and after adding exclusive channels

As a part of the process of replacing a channel by multiple channels, transformation $T_{ex}$ modifies the state machines that communicate over these channels. Before transformation, both state machines that are part of object *a* send signals via port *Out*, and both state machines that are part of object *b* receive signals via port *In*. After transformation, one of the state machine of object *a* sends signals via ports *Out1* and *Out3*, and the other uses ports *Out2* and *Out4*. The receiving state machines are modified in a similar fashion. Figure 3.8 shows parts of one of the state machine of object *a* and parts of one of the state machines of object *b* before and after transformation. The names of the states of these partial state machines correspond to the names of the classes they belong to. The situation before transformation is illustrated on the left of the figure, and the situation after transformation is shown on the right.

### Reducing the Number of Channels

When two objects are connected by more than one channel, these channels can be merged into one if they have the same type and directionality, and support the same argument types. Therefore, we implemented a transformation $T_{mc}$ that merges multiple channels between a pair of objects into one channel. Merging channels is a way of optimizing models because it can be used to reduce the number of instances of the CABP that need to be added.

### Cloning Classes

Many of the transformations described above use two auxiliary transformations. One of these transformations takes a model and a channel as input, and clones the classes of the objects that communicate over this channel. After applying this transformation, the objects that communicate over the channel are instances of the new cloned classes, and all remaining instances of the original classes remain unchanged. This transformation ensures that all transformations that alter objects that communicate over a certain channel only affect these particular objects.

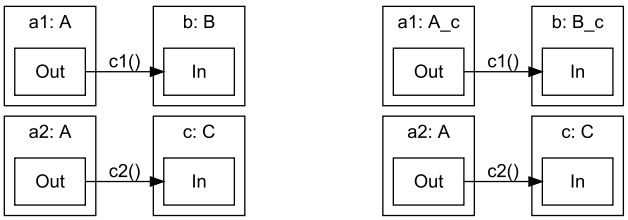


Figure 3.9: Communication diagrams of a model before and after cloning classes

Figure 3.9 shows the communication diagram of a model before and after applying this transformation. After transformation, the classes of the objects communicating over channel *c1* are cloned. In the resulting model, object *a1* is an instance of class $A\_c$ and object $b$ an instance of class $B\_c$.

### Removing Unused Classes

The second auxiliary transformation used by the transformations described above removes all uninstantiated classes from a model. The model depicted on the right of Figure 3.9, for example, no longer contains an instance of class $B$, which means that this class can be removed without affecting the system specified by the model.

## 3.2.2 Exogenous Transformations

Each of the following exogenous model transformations takes an SLCO model as input and produces a model or an implementation in one of the target languages. Because of the gaps described in Section 3.1, the SLCO model provided as input must be specified using a subset of SLCO that matches the capabilities of the target language and platform. Any model that is specified using constructs that have no direct counterparts in the target language must first be refined using the transformations described in Section 3.2.1.

### Transforming SLCO to POOSL

Because of the gaps between SLCO and POOSL, the transformation from SLCO to POOSL is restricted to models that contain only synchronous channels. An example of the output of this transformation is shown in Listing 3.1. This fragment of a POOSL model is the result of applying the transformation to a slightly modified version of the model of Figures 1.5, 1.6, and 1.8. The model has been modified by replacing all asynchronous channels by synchronous channels.

The transformation transforms each SLCO class to a POOSL process class. Lines 2 to 16 of Listing 3.1 show the process class that represents the SLCO class $P$. The state machines of each class are transformed to a number of process methods, one for each state. The method in lines 11 to 13 represents the state *Rec1*, the method in lines 14 to 16 represents the state *SendRec*, and the

Listing 3.1: Part of a POOSL model

```
1  ...
2  process class P()
3    instance variables m: Integer, PSendRecs: String
4    communication channels In1, In2, InOut
5    message interface InOut?T(String); In2?Q(Integer); ...
6    initial method call P_initial()()
7    instance methods
8      P_initial()() | |
9        m := 0; par Rec1_Rec1()() and Rec2_Rec2()() and SendRec_SendRec()() rap
10     .
11     Rec1_Rec1()() | var_1: Boolean |
12       In1?P(var_1 | var_1 = false); Rec1_Rec1()()
13     .
14     SendRec_SendRec()() | |
15       [m = 6] skip; InOut!S("a"); InOut?T(PSendRecs); SendRec_SendRec()()
16     .
17 ...
18     Com_Com0()() | |
19       sel delay(5) or Out1!P(true); Com_Com1()() les
20     .
21 ...
22 behaviour specification
23   (p: P[c3/InOut, c1/In1, c2/In2] || q: Q[c1/Out1, c2/Out2, c3/InOut])
24   \ {c1, c2, c3}
25 .
```

method in lines 18 to 20 represents the state *Com0*. Besides this, an additional process method is generated for each class, which calls all process methods representing the initial states of the state machines of the class. These additional process methods also initialize the variables of the classes and state machines to their initial values, if applicable. Lines 8 to 10 show the process method that initializes the variable $m$ of class $P$ and calls the process methods that represent the initial states of its state machines as part of a parallel composition.

Since each SLCO class corresponds to exactly one POOSL process class, it is clear that the global variables of an SLCO class can be represented by variables of the corresponding POOSL class. An SLCO state machine, however, is represented by multiple process methods, and a group of process methods within a process class cannot share variables that are inaccessible by other process methods of that class. For this reason, the local variables of SLCO state machines have to be represented by variables of process classes too. These variables can no longer be considered to be local to a certain state machine, since all the process methods of a class can access them, even those representing other state machines of the same class. Line 3 shows the declaration of the POOSL variables representing the global SLCO variable $m$ and the local SLCO variable $s$ of state machine *SendRec*.

States with a single outgoing transition are translated to process methods containing a single sequence of statements. This sequence of statements represents the outgoing transition. Line 12 shows the sequence of statements representing the outgoing transition of state *Rec1*, and line 15 shows the sequence of statements representing the outgoing transition of state *SendRec*.

States with multiple outgoing transitions are translated to process methods containing a select statement. Each of the alternatives of such a select statement represents one of the outgoing transitions. Line 19 shows the select statement representing the outgoing transitions of state *Com0*. The semantics of the POOSL select statement is as follows. Each of the alternatives of a select statement is a sequence of statements. If one or more of these sequences starts with a statement

that is enabled, one of these sequences 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.

As mentioned above, each transition is transformed to a sequence of statements. If a transition leads to an ordinary state, the sequence of statements representing this transition ends with a process method call. This process method call calls the method representing the target state of the transition. Line 12 shows the sequence of statements that represents the transition from state *Rec1* to itself. If a transition leads to a final state, this process method call is omitted. The first alternative of the select statement on line 19 shows an example of this situation. After 5 ms, the method representing state *Com0* terminates.

POOSL does not offer the form of conditional signal reception that uses expressions as arguments to limit the types of signals that can be received. For this reason, auxiliary variables are introduced to mimic this form of conditional signal reception. Lines 11 and 12 show how variable *var_1* is used to do so.

Lines 23 and 24 show that objects $p$ and $q$ are declared to be instances of classes $P$ and $Q$, and that channels *c1*, *c2*, and *c3* are connected to the ports of these objects.

### Transforming SLCO to NQC

The transformation from SLCO to NQC transforms SLCO models to NQC implementations, provided that the following two conditions hold. First, all objects describing the behavior of the controllers must communicate with each other via asynchronous, lossy channels. Second, all communication between these objects and the object(s) representing the hardware environment takes place over synchronous channels.

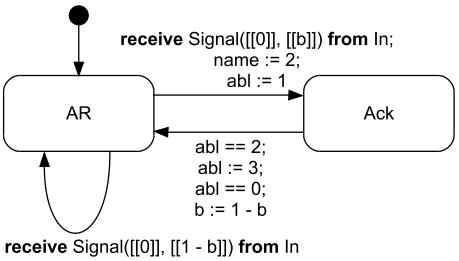


Figure 3.10: Part of the Concurrent Alternating Bit Protocol used to illustrate the transformation from SLCO to NQC

Figure 3.10 shows the state machine of object *ar* that is part of the implementation of the CABP introduced in Section 3.2.1 after merging it with objects *a* and *sender*. Listing 3.2 shows a fragment of NQC code that is the result of applying the transformation from SLCO to NQC to a model containing the merged objects.

Every state machine that describes the behavior of an object in an SLCO model is transformed into an NQC task. Lines 2 to 23 show the task that represents the state machine of Figure 3.10. Line 3 shows the declaration and initialization of local variable *b* and auxiliary variable *temp*. The purpose of auxiliary variable *temp* is explained below. Global variables of SLCO objects, such as *abl* and *name* in Figure 3.10, are represented by global variables of NQC programs, and are not shown in Listing 3.2.

Each state is represented by a label, followed by sequences of statements that represent the outgoing transitions of the state, and a goto statement that jumps back to the label. Lines 4

Listing 3.2: Fragment of NQC code

```
1  ...
2  task AR() {
3    int b = 0; int temp;
4    AR:
5      temp = Message();
6      if (
7        temp != 0 && ((temp & 112) / 16) == 0 && ((temp & 128) / 128) == b
8      ) {
9        name = 2; abl = 1; goto Ack;
10     }
11     temp = Message();
12     if (
13       temp != 0 && ((temp & 112) / 16) == 0 && ((temp & 128) / 128) == (1 - b)
14     ) {
15       goto AR;
16     }
17   goto AR;
18   Ack:
19     if (abl == 2) {
20       abl = 3; /* skip */; until (abl == 0); b = 1 - b; goto AR;
21     }
22   goto Ack;
23 }
```

to 17 show the labeled sequence of statements representing state $AR$, and lines 18 to 22 represent state $Ack$.

If the statements of an SLCO transition start with a statement that might block execution, this statement is translated to an NQC if statement and the remaining SLCO statements are translated to NQC statements that form the body of this if statement. In case the first statement is a signal reception statement, the if statement is preceded by a call to the API function *Message*. The purpose of this API call is explained below. Lines 5 to 10, 11 to 16, and 19 to 21 represent the transitions of the state machine in Figure 3.10. The remaining statements that are part of an SLCO transition are translated as follows. Assignments in SLCO are translated to assignments in NQC, as shown in lines 9 and 20. Expressions and signal receptions are translated to until statements, as shown in line 20. Statements that send signals are not shown in this example. They are translated to calls to the API function *SendMessage*, using the encoding explained below.

An RCX controller can only send and receive integer values over its infrared port. Since signals in SLCO consist of a name and possibly a number of arguments, they have to be encoded such that they can be represented as a single integer before they can be sent, and they have to be decoded after being received. Lines 5 and 11 show that first the auxiliary variable *temp* is used to store the value of the integer that has been received last. Then, this integer is decoded as shown in lines 7 and 13. The encoding of a signal, which is not shown in this example, is done using a similar procedure. The actual sending and receiving of integer values is done via API calls to the functions *Message* and *SendMessage*, as mentioned above.

In addition to an SLCO model, the transformation from SLCO to NQC also takes a mapping from elements in the SLCO model to concepts of NQC as input. Listing 3.3 shows a part of the mapping for the model described in Section 3.3. For each class that needs to be transformed to an NQC program, this mapping defines how some of the signals of an SLCO model correspond to information received from sensors and commands sent to motors. Each signal that is not mentioned in this mapping is assumed to be part of the communication between controllers via infrared and is translated accordingly, as described above.

Listing 3.3: Part of a mapping from SLCO to NQC

```
1  class Middle
2    port2motor
3      Motor -> OutB 7
4    port2sensor
5      Sensor -> Sensor2 Light
6    signal2motor
7      Motor Left -> On Reverse
8      Motor Right -> On Forward
9      Motor Off -> Float
10   signal2sensor
11     Sensor Block -> Below -10
12     Sensor BlockPassed -> Above -2
```

In line 3, port *Motor* of an SLCO model is mapped to output port *OutB* of a Lego Mindstorms controller. The speed of the motor is set to 7. In line 5, SLCO port *Sensor* is mapped to input port *Sensor2* of a Lego Mindstorms controller, which is connected to a light sensor. Signals named *Left*, *Right*, and *Middle* sent over port *Motor* are mapped to commands for a motor in lines 7 to 9. Lines 11 and 12 show how signals named *Block* and *BlockPassed* are mapped to information received from sensors. In line 11, for example, receiving a signal named *Block* is mapped to the event that the value transmitted by the sensor connected to port *Sensor2* drops below $-10$.

### Transforming SLCO to Promela

The transformation from SLCO to Promela transforms SLCO models containing only synchronous channels and asynchronous, lossless channels to Promela models. Listing 3.4 shows a fragment of a Promela model that is the result of applying this transformation to a slightly modified version of the SLCO model described in Figures 1.5, 1.6, and 1.8. Because Promela does not support the more general form of conditional signal reception, the condition $m >= 0$ is removed from the signal reception in the SLCO model. Additionally, the asynchronous, lossy channel is replaced by an asynchronous, lossless channel.

Listing 3.4: Part of a Promela model

```
1  mtype {S, P, Q, T, a}
2  int p_m = 0
3  chan c1_q2p = [1] of {mtype, bool}
4  chan c3_1_q2p = [0] of {mtype, mtype}
5  chan c3_2_q2p = [0] of {mtype, mtype}
6  ...
7  active [1] proctype p_Rec1() {
8    Label_Rec1: {
9      if :: {c1_q2p?P,eval(false); goto Label_Rec1} fi
10   }
11 }
12
13 active [1] proctype p_Rec2() {
14   Label_Rec2: {
15     if :: {c2_q2p?Q,p_m; p_m = p_m + 1; goto Label_Rec2} fi
16   }
17 }
18 ...
19 active [1] proctype q_Com() {
20   mtype q_s;
21   Label_Com0: {
22     if :: {skip; goto Label_Com2}
23        :: {c1_q2p!P,true; goto Label_Com1}
24     fi
25   }
26   Label_Com1: {
27     if ::
28       {c2_q2p!Q,5; c3_1_p2q?S,q_s; c3_2_q2p!T,q_s; goto Label_Com2}
29     fi
30   }
31   Label_Com2: skip
32 }
```

For each object in an SLCO model, the state machines that describe its behavior are transformed to Promela processes. Lines 7 to 11 in Listing 3.4 show the process that represents state machine *Rec1* of object $p$, lines 13 to 17 show the process that represents state machine *Rec2* of object $p$, and lines 19 to 32 show the process that represents state machine *Com* of object $q$.

Channels between objects in SLCO are transformed to channels between processes in Promela. Line 3 shows the declaration of the Promela channel representing the SLCO channel named *c1*. The 1 between square brackets indicates that this channel is asynchronous. The declaration specifies that this channel is suited for messages consisting of two parts: a symbolic name (*mtype*) and a Boolean value. Lines 4 and 5 show the declaration of two channels representing the SLCO channel named *c3*. Although Promela offers bidirectional channels, a separate channel is declared for each direction. This is done because the semantics of channels in Promela differs from their semantics in SLCO. In SLCO, an object can only receive signals sent by another object. In Promela, however, a process can retrieve a message from a channel that it has sent over this channel itself. For this reason, we introduce two Promela channels for each bidirectional SLCO channel. Each of the Promela channels is used only to communicate in one direction, which means that a process uses the channel either to send messages or to receive messages, but not both. The 0 between square brackets indicates that this channel is synchronous.

The local variables of state machines are represented by local variables of processes, as shown in line 20. Global variables of SLCO objects are represented by global variables of the Promela model. They are accessible by all processes in the model. On line 2, global variable $m$ of object $p$ is declared.

Ordinary states are transformed to labeled selection statements, and final states are transformed to labeled skip statements. Lines 21 to 25 represent state *Com0* of object $q$, lines 26 to 30 represent state *Com1*, and line 31 represents state *Com2*.

Every outgoing transition of a state is represented by an alternative of the selection statement that represents this state. Each of these alternatives ends with a goto statement to the label representing the target state of the transition. State *Rec1* of object $p$ has only one outgoing transition, as shown in line 9. State *Com0* of object $q$ has two outgoing transitions. The transition to state *Com2* is shown in line 22, and the transition to state *Com1* is shown in line 23.

The semantics of the selection statement is such that it will non-deterministically execute one of the alternatives for which the first statement is executable, and it will block if none of these statements are executable. The statements of a transition are transformed to Promela statements in a straightforward way. Expressions and assignments in SLCO are translated to equivalent expressions and assignments in Promela, as shown in line 15. A signal reception is transformed to a receive statement, as shown in line 9. The receive statement $c1\_q2p?P, \mathbf{eval}(\mathbf{false})$ represents the reception of signals named $P$ with an argument that must be equal to **false**. A Promela receive statement blocks until it is able to receive a message over a channel. A send signal statement is transformed to a send statement in a similar fashion, as shown in lines 23 and 28.

## 3.3 Sequences of Transformations

To illustrate how the transformations described in Section 3.2 can be applied, we introduce a system of three cooperating conveyor belts, schematically depicted in Figure 3.11. We show how different implementations for controlling this system can be generated by composing a number of transformations into different sequences of transformations. The two vertical rectangles in Figure 3.11 denote conveyor belts that transport items towards another conveyor belt, which is represented by a horizontal rectangle. The arrows in the rectangles denote the directions in which these conveyor belts can transport items. The three belts should cooperate such that items supplied by the vertical belts are dropped onto the horizontal belt, one by one. The horizontal

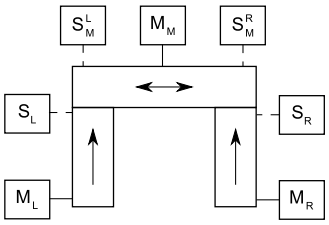


Figure 3.11: Cooperating conveyor belts

belt should transport each of these items from the right-most belt to the left and those from the left-most belt to the right. The small rectangles labeled $M_L$, $M_M$, and $M_R$ depict the motors that drive the conveyor belts, and the small rectangles labeled $S_L$, $S_M^L$, $S_M^R$, and $S_R$ depict the sensors that detect the passing items.

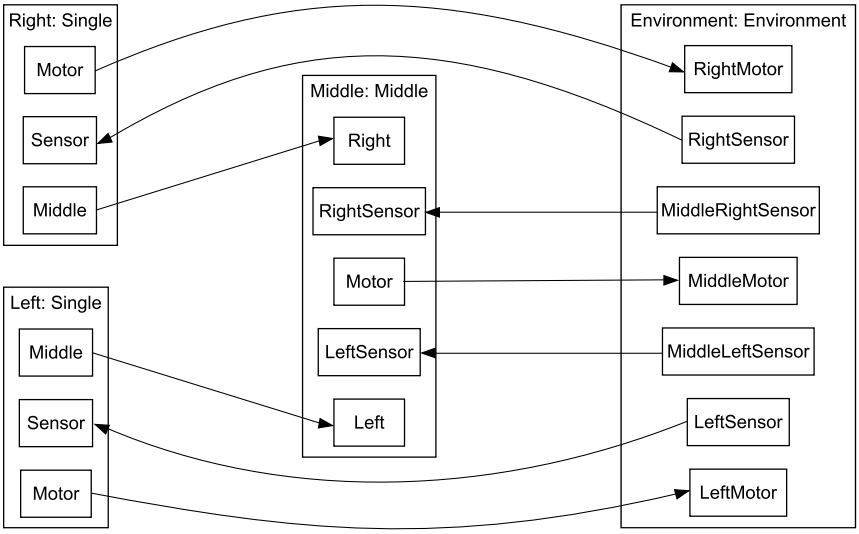


Figure 3.12: Communication between the controllers of the cooperating conveyor belts and their environment

An SLCO model that describes this system consists of four components: two objects (*Left* and *Right*) that model the controllers for the vertical belts, one object (*Middle*) that models the controller for the horizontal belt, and one object (*Environment*) that models the environment. The communication diagram in Figure 3.12 shows how these objects are connected. Among other things, it also shows that the objects *Left* and *Right* are both instances of class *Single*. To increase the readability of the figure, the names of the channels have been omitted.

Figure 3.13 shows the two state machines that specify the behavior of the controllers of this system. The state machine on the left specifies the behavior of objects *Left* and *Right*, and the state machine on the right specifies the behavior of object *Middle*. The state machines related to the environment are not shown.

In Figure 3.14, three sequences of model transformations leading to three distinct implementations with the same observable behavior are shown. In this figure, SLCO models are represented by the rectangles labeled $M^1_{SLCO}$ to $M^{16}_{SLCO}$, NQC implementations by rectangles labeled $M^1_{NQC}$ to $M^3_{NQC}$, single transformations by labeled, solid arrows, and subsequences of transformation by

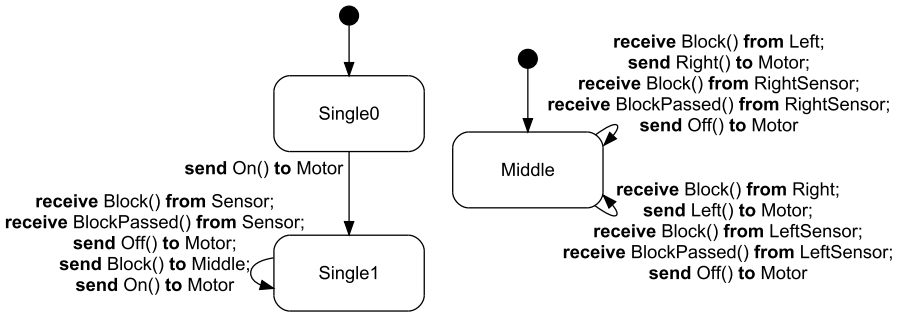


Figure 3.13: Behavior of the controllers of the cooperating conveyor belts

labeled, dashed arrows. The starting point of all three sequences, model $M^1_{SLCO}$, describes the intended cooperation in terms of objects *Left*, *Middle*, *Right*, and their hardware environment.

The topmost sequence of transformations in Figure 3.14 transforms the SLCO model $M^1_{SLCO}$ to the NQC implementation $M^2_{NQC}$, which is meant to be deployed on two controllers. First, objects *Left* and *Right* are merged, and the channels that connect these objects to *Middle* are also merged. To be able to distinguish the origin of signals after merging the channels, transformation $T_{ic}$ is applied before merging the channels. Then, synchronous communication is replaced by asynchronous communication, after which lossless communication over a lossy channel is achieved by adding objects that implement the CABP. Because this last transformation adds a number of objects to the model and the resulting model $M^8_{SLCO}$ should only contain as many objects as there are controllers, these objects must be merged with others to reduce the total number of objects again. As mentioned above, objects can only be merged if all pairs of state machines involved in communication communicate over distinct channels. This explains why transformation $T_{ex}$ is applied before transformation $T_{merge}$ is used to reduce the number of objects. In the final transformation step, all string constants are replaced by integer constants, leading to model $M^8_{SLCO}$. Because all semantic gaps and platform gaps have been bridged, transforming this model into implementation $M^2_{NQC}$ using transformation $T_{NQC}$ is straightforward.

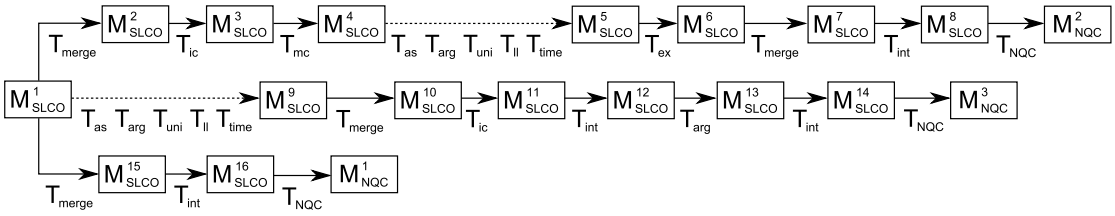


Figure 3.14: Sequences of transformations that transform a single model to three implementations

The sequence of transformations in the middle of Figure 3.14 leads to an implementation with three controllers. Because the three controllers communicate by broadcasting signals, the origin of signals has to be made explicit using transformation $T_{ic}$. After applying this transformations, the names of the signals have to be changed again, and all string constant have to be replaced by integer constants, to be able to apply transformation $T_{NQC}$. Transformation $T_{NQC}$ encodes the arguments of signals into a single integer, as discussed in Section 3.2.2. To keep the number of distinct integer constants as low as possible and achieve an optimal encoding of the signal arguments, transformation $T_{int}$ is applied both before and after transformation $T_{arg}$.

The sequence of transformations in the bottom of Figure 3.14 generates an implementation for a system with only one controller. Therefore, all objects are merged in the first transformation step.

# Chapter 4

# Implementation

In this chapter, we discuss the implementation of all languages and model transformations related to SLCO, and we describe the software tools, languages, and platforms used to implement them. All the grammars, metamodels, and transformations described below are available online[1].

## 4.1 Implementation

Figure 4.1 gives an overview of the languages and transformations related to SLCO. In this figure, all languages that are based on a metamodel implemented using the Eclipse Modeling Framework (EMF) [28] are depicted as rectangles. The rounded rectangles represent languages that have a textual concrete syntax, and the arrows represent model transformations, parsers, and template-based code generators, which either transform models or convert them from one representation to another.

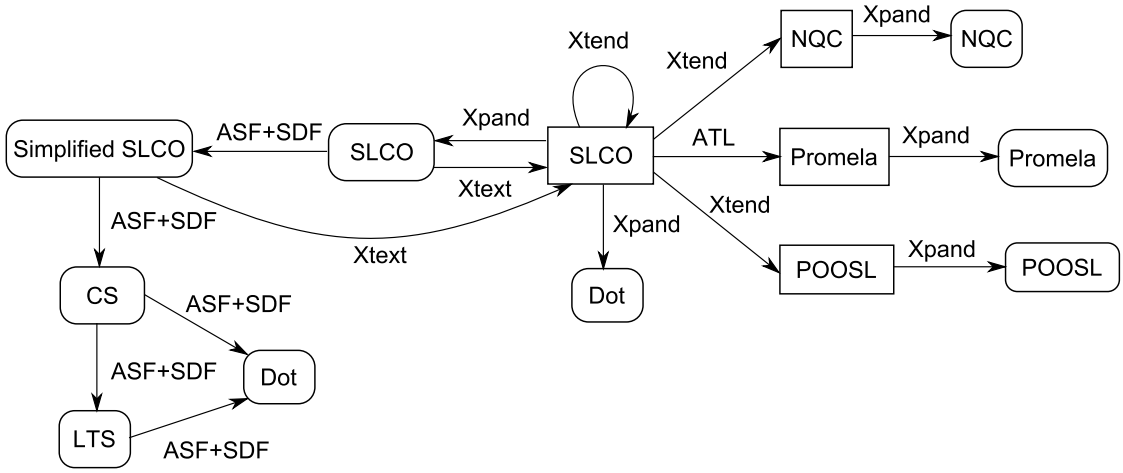


Figure 4.1: Overview of the languages and transformations related to SLCO

The metamodels of SLCO, NQC, Promela, and POOSL define the abstract syntax of these languages. They define the concepts offered by the languages and the relations between these

[1] http://code.google.com/p/simple-language-of-communicating-objects/

concepts. To enable creation of models, EMF offers the automatic generation of a tree-view editor from metamodels.

Unfortunately, creating large models using the standard editor provided by EMF is cumbersome. To ease the process of modeling, we defined a textual syntax for SLCO with Xtext [14], which also provides us with a textual editor for SLCO. Since all other models are automatically generated from SLCO models, there is no need for convenient editors for the other languages involved.

All endogenous transformations that are used to bridge the gaps between SLCO, NQC, Promela, and POOSL are implemented using the Xtend model transformation formalism [17]. In Figure 4.1, these transformations are represented by the arrow that connects the rectangle labeled SLCO to itself. The transformations from SLCO to POOSL and NQC are also implemented using Xtend, and the transformation from SLCO to Promela is implemented using the ATL Transformation Language [21].

The result of each of these transformations is a model that conforms to the corresponding metamodel. These models cannot be used directly in POOSL and Spin or on the Lego Mindstorms platform. Instead, models in textual form are required for simulation, verification, and execution. Therefore, we implemented model-to-text transformations for these tools using Xpand [17]. Additionally, Xpand is used to produce the diagrams that form the graphical syntax of SLCO. Each of the diagrams is in fact a directed graph written in Dot that can be visualized with the Graphviz tool [15].

The textual languages CS and LTS, their relation to Dot, and the transformations implemented in ASF+SDF [10] are discussed in detail by Andova et al. [4] and in the PhD thesis of Luc Engelen [16]. They also describe the relation between these languages and the basic version of SLCO discussed in Sections 1.1 and 1.2. Because every textual model that is expressible in the simplified version of SLCO is also expressible in the regular version of the language, the parser and editor that are generated from the Xtext grammar of SLCO are also suited for the manipulation of simplified textual SLCO models.

## 4.2 ASF+SDF and the Meta-Environment

The language ASF+SDF [13] is a combination of the two formalisms ASF [9] and SDF [30]. The Syntax Definition Formalism (SDF) is a formalism for the definition of the syntax of context-free languages. The Algebraic Specification Formalism (ASF) is a formalism for the definition of conditional rewrite rules. Given a syntax definition in SDF of a source and target language, ASF can be used to define a transformation from the source language to the target language. In ASF, conditional rewrite rules are specified using the concrete syntax of the input and output languages.

Context-free languages are closed under union and, as a result of this, the SDF definitions of two languages can be combined to form the definition of a new context-free language, without altering the existing definitions. However, because ambiguities may arise after combining syntax definitions, it might be necessary to add constructs for disambiguation to a definition that combines existing languages. Using ASF in combination with SDF to implement transformations guarantees syntax safety. A transformation is syntax safe if it only accepts input that adheres to the syntax definition of the input language and always produces output adhering to the definition of the output language.

The ASF+SDF Meta-Environment [10] is an integrated development environment (IDE) for ASF+SDF. It has a graphical user interface that offers syntax-highlighting for the specification of SDF and ASF definitions, and an interpreter and debugger for the execution and debugging of ASF specifications. It can be used to create a command-line tool that parses and rewrites input adhering to the syntax definition of the input language and outputs the result. These command-line tools employ memoization, which ensures that the result of a rewrite rule applied to a given term

is computed only once. Both the ASF+SDF Meta-Environment and the command-line tools it generates use Annotated Terms (ATerms) [11] to represent terms internally. Because ATerms offer maximal subterm sharing, the internal representation of terms uses as little space as possible.

Listing 4.1: Part of an SDF definition that defines simple Boolean expressions

```
1  sorts
2    BoolCon BoolExp
3  context-free syntax
4    "true" | "false" -> BoolCon
5    BoolCon -> BoolExp
6    BoolExp "xor" BoolExp -> BoolExp {right}
7    eval(BoolExp) -> BoolCon
8  variables
9    "$BoolCon"[0-9]* -> BoolCon
10   "$BoolExp"[0-9]* -> BoolExp
```

Listing 4.1 shows a part of an SDF definition that defines a syntax for simple Boolean expressions. In ASF+SDF, each term conforms to a sort. On line 2, two such sorts are introduced, whose syntax is defined in lines 4 to 7. Line 4 states that a term of sort *BoolCon* is of the form "true" or "false". A term of sort *BoolCon* is also a valid term of sort *BoolExp*, as defined in line 5. Furthermore, line 6 specifies that two terms of sort *BoolCon* joined by the right-associative operator "xor" form a term of sort *BoolExp*. The signature of an evaluation function for Boolean expressions is specified on line 7. On lines 9 and 10, variables that represent terms of the aforementioned sorts are introduced.

Listing 4.2: ASF rule for the evaluation of simple Boolean expressions

```
1  [rule0]
2    eval($BoolCon) = $BoolCon
3
4  [rule1]
5    eval($BoolCon xor $BoolCon) = false
6
7  [rule2]
8    $BoolCon1 != $BoolCon2
9    ====>
10   eval($BoolCon1 xor $BoolCon2) = true
11
12 [default-rule]
13   $BoolCon1 := eval($BoolExp1),
14   $BoolCon2 := eval($BoolExp2)
15   ====>
16   eval($BoolExp1 xor $BoolExp2) = eval($BoolCon1 xor $BoolCon2)
```

Listing 4.2 shows the (conditional) rewrite rules that define how the simple Boolean expressions of Listing 4.1 are evaluated. The first part of an ASF rule is optional and consists of the conditions of the rule, which are separated from the rest of the rule by an arrow (====>). Next, the left-hand side and right-hand side of the rule follow, separated by an equality sign. If a rule has no conditions or all its conditions hold, its application to a term results in replacing the left-hand side by the right-hand side. ASF+SDF offers two kinds of conditions: (in)equality conditions and matching conditions. An equality condition consists of a right-hand side and a left-hand side, separated by two equal signs (==). The condition holds if both sides can be matched. The right-hand side and left-hand side of an inequality condition are separated by an exclamation mark and an equal sign (!=), and it holds if both sides cannot be matched. Similarly, a matching condition consists

of a right-hand side and a left-hand side, separated by a colon and an equal sign (`:=`). Also this type of condition holds if both sides can be matched. In this case, however, if both sides can be matched, the variables occurring at the left-hand side are instantiated accordingly. In Listing 4.2, the first two rules have no conditions. The inequality condition of the third rule is shown on line 8, and the matching conditions of the last rule are shown on lines 13 and 14. On lines 1, 4, 7, and 12, the identifiers of the rules are shown. The last rule is only applied if non of the other rules are applicable, which is indicated by the fact that its identifier starts with "default".

## 4.3 openArchitectureWare

The openArchitectureWare platform offers a number of tools related to model transformation: Xpand is used for model-to-text transformations, Xtext [14] is used for text-to-model transformations, and Xtend is used for model-to-model transformations. Here, the term "model" refers to an instance of an explicit metamodel. Execution of model-to-text transformations implemented with Xpand and model-to-model transformations implemented with Xtend can be automated using scripts for the Modeling Workflow Engine for Eclipse. Xpand and Xtend are based on the same type system and expression language. The type system offers simple types, such as string, Boolean, and integer, collection types, such as list and set, and the possibility to import metamodels. The expression language offers a number of basic constructs that can be used to create expressions, such as literals, operators, quantifiers, and switch expressions.

Currently, the platform no longer exists on its own and has become a part of the Eclipse Modeling Project[2] instead. It is implemented as a number of Eclipse plug-ins and is based on the Eclipse Modeling Framework (EMF) [28].

### 4.3.1 Xpand

Xpand is a template-based language that generates text files given an EMF model. An Xpand template takes a metaclass and a list of parameters as input and produces output by executing a list of statements. There are a number of different types of statements, including one that saves the output generated by its statements to a file and one that triggers the execution of another template.

### 4.3.2 Xtext

Xtext is a tool that parses text and converts it to an equivalent model, given a grammar describing the syntax of the input. Xtext uses ANTLR [25] to generate a parser that parses the textual representations of models. An Xtext specification consists of rules that define both a metamodel and a mapping from concrete syntax to this metamodel. Given a grammar, Xtext also generates an editor that provides features such as syntax highlighting and code completion.

### 4.3.3 Xtend

Xtend is a functional language for model transformation. It adds extensions to the basic expression language, which take a number of parameters as input and return the result of an expression. Because the extensions are not side-effect free, Xtend is not a pure functional language. Transformations implemented in Xtend are unidirectional, which means that they can only be used to transform models in a single direction. In other words, a transformation for given source and target metamodels can transform models conforming to the source metamodel into models

[2] `http://www.eclipse.org/modeling/`

conforming to the target metamodel, but not the other way around. The language can be used for in-place transformations, which modify a given model, as well as transformations that produce new models. Xtend is an interpreted language. It is supported by an IDE offering syntax highlighting, debugging, and code completion.

## 4.4 ATL Transformation Language

The ATL Transformation Language (ATL) [20], previously known as the ATLAS transformation language, is another EMF based language for model transformation. Similarly to Xtend, ATL is also a unidirectional transformations language that offers both in-place transformations and creation of new models. The language provides both declarative and imperative constructs for the definition of model transformations. In contrast to Xtend, ATL does not have a native syntax for expressions, but uses the Object Constraint Language (OCL) [24] instead. ATL is supported by an IDE that offers debugging and syntax highlighting. A virtual machine is used to execute transformations after translating them to byte code. The execution of ATL transformations can be automated using ant tasks, which are small scripts that make it possible, for example, to compose transformations with and without saving the intermediate models.

## 4.5 Dot and Graphviz

Dot is a language for graph visualization that is part of the Graphviz toolset [15]. Given a description of a graph written in Dot, Graphviz can visualize this graph as an image in various output formats. Graphviz employs layout algorithms to achieve optimal placement of nodes and edges. A graph description in Dot is a list of nodes and edges from node to node, combined with attributes that specify how particular nodes and edges should be displayed. These attributes define, for example, the color, width, height, and the type of lines used to draw these nodes and edges.

Although Dot is not designed specifically for applications in model-driven software engineering, we use it extensively for the visualization of graphical diagrams representing models. Currently, EMF based alternatives for graphical modeling do not provide the functionality required to create such diagrams.

# Bibliography


[1] M.F. van Amstel, M.G.J. van den Brand, and L.J.P. Engelen. An Exercise in Iterative Domain-Specific Language Design. In *Proceedings of the Joint ERCIM Workshop on Software Evolution and International Workshop on Principles of Software Evolution*, 2010. doi:10.1145/1862372.1862386.

[2] M.F. van Amstel, M.G.J. van den Brand, and L.J.P. Engelen. Using a DSL and Fine-Grained Model Transformations to Explore the Boundaries of Model Verification. In *Proceedings of the 3rd Workshop on Model-Based Verification and Validation*, 2011. doi:10.1109/SSIRI-C.2011.26.

[3] 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 1st International Conference on Model Transformation*, 2008. doi:10.1007/978-3-540-69927-9_5.

[4] S. Andova, M.G.J. van den Brand, and L.J.P. Engelen. Prototyping the Semantics of a DSL using ASF+SDF: Link to Formal Verification of DSL Models. In *Proceedings of the 2nd International Workshop on Algebraic Methods in Model-based Software Engineering*, 2011. doi:10.4204/EPTCS.56.5.

[5] S. Andova, M.G.J. van den Brand, and L.J.P. Engelen. Reusable and Correct Endogenous Model Transformations. In *Proceedings of the 5th International Conference on Model Transformation*, 2012. doi:10.1007/978-3-642-30476-7_5.

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

[7] K.A. Bartlett, R.A. Scantlebury, and P.T. Wilkinson. A Note on Reliable Full-Duplex Transmission over Half-Duplex Links. *Communications of the ACM*, 1969. doi:10.1145/362946.362970.

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

[9] J.A. Bergstra. *Algebraic Specification*, chapter 1. ACM, 1989.

[10] M.G.J. van den Brand, J. Heering, H. de Jong, M. de Jonge, T. Kuipers, P. Klint, L. Moonen, P. Olivier, J. Scheerder, J.J. Vinju, E. Visser, and J. Visser. The ASF+SDF Meta-Environment: a Component-Based Language Development Environment. In *Proceedings of the 10th International Conference on Compiler Construction*, 2001. doi:10.1007/3-540-45306-7_26.

[11] M.G.J. van den Brand and P. Klint. ATerms for Manipulation and Exchange of Structured Data: It's All About Sharing. *Information and Software Technology*, 2007. doi:10.1016/j.infsof.2006.08.009.

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

[13] A. van Deursen. An overview of ASF+SDF. In *Language Prototyping: An Algebraic Specification Approach*. World Scientific Publishing Co., 1996.

[14] S. Efftinge and M. Völter. oAW xText: a Framework for Textual DSLs. In *Proceedings of the Modeling Symposium at Eclipse Summit*, 2006.

[15] J. Ellson, E. Gansner, L. Koutsofios, S. North, and G. Woodhull. Graphviz – Open Source Graph Drawing Tools. In *International Symposium on Graph Drawing*, 2002. doi:10.1007/3-540-45848-4_57.

[16] L.J.P. Engelen. *From Napkin Sketches to Reliable Software*. PhD thesis, Eindhoven University of Technology, 2012.

[17] 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.

[18] M. Hennessy. *The Semantics of Programming Languages*. Wiley, 1990.

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

[20] F. Jouault, F. Allilaire, J. Bézivin, I. Kurtev, and P. Valduriez. ATL: a QVT-like Transformation Language. In *Companion to the 21st ACM SIGPLAN Symposium on Object-Oriented Programming Systems, Languages, and Applications*, 2006. doi:10.1145/1176617.1176691.

[21] F. Jouault and I. Kurtev. Transforming Models with ATL. In *MoDELS 2005 Satellite Events*, 2005. doi:10.1007/11663430_14.

[22] T. Mens and P. van Gorp. A Taxonomy of Model Transformation. *Electronic Notes in Theoretical Computer Science*, 2006. doi:10.1016/j.entcs.2005.10.021.

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

[24] Object Management Group. Object Constraint Language, Version 2.3.1, January 2012.

[25] T.J. Parr and R.W. Quong. ANTLR: A Predicated-LL(k) Parser Generator. *Software — Practice and Experience*, 1995. doi:10.1002/spe.4380250705.

[26] G.D. Plotkin. A Structural Approach to Operational Semantics. *Journal of Logic and Algebraic Programming*, 2004. doi:10.1016/j.jlap.2004.05.001.

[27] A. Pnueli. The Temporal Logic of Programs. In *Proceedings fo the 18th Annual Symposium on Foundations of Computer Science*, 1977. doi:10.1109/SFCS.1977.32.

[28] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks. *EMF: Eclipse Modeling Framework*. Addison-Wesley Professional, 2008.

[29] 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 IEEE/ACM International Conference on Formal Methods and Models for Codesign*, 2007. doi:10.1109/MEMCOD.2007.371231.

[30] E. Visser. *Syntax Definition for Language Prototyping*. PhD thesis, University of Amsterdam, 1997.

# Appendix A

# Operational Semantics of SLCO

## A.1 Introduction

This appendix discusses the formal operational semantics of SLCO. However, successful termination and time are not taken into account. We start with a description of the syntax of SLCO, followed by a description of the rules that define its operational semantics. Finally, we discuss the initialization of the evaluation functions that are used in these rules.

## A.2 Syntax

In this section, we use a variant of EBNF to define the syntax of SLCO. Although the use of quotation marks may suggest otherwise, the following syntax does not qualify as a concrete syntax for the language, because it does not assign a unique parse tree to each fragment of the language. Instead, the quotation marks are used to distinguish EBNF symbols from symbols of SLCO. For each element $e$ of a syntactic category, zero or more occurrences of $e$ are denoted by $e^*$, one or more occurrences are denoted by $e^+$, and zero or one occurrence is denoted by $[e]$.

The syntax of models $m \in \mathit{Models}$, classes $\mathit{class}^* \in \mathit{Classes}$, objects $\mathit{obj} \in \mathit{Objects}$, channels $\mathit{chan} \in \mathit{Channels}$, and variables $\mathit{var} \in \mathit{Variables}$ is defined as follows:

$$
\begin{array}{lcl}
m & ::= & mn\ class^*\ obj^*\ chan^* \\
class & ::= & cn\ var^*\ pn^*\ sm^* \\
obj & ::= & on\ \text{“:”}\ cn \\
chan & ::= & chn\ \text{“(”}\ type^*\ \text{“)”}\ chtype\ \textbf{“from”}\ on\ \text{“.”}\ pn\ \textbf{“to”}\ on'\ \text{“.”}\ pn' \\
 & | & chn\ \text{“(”}\ type^*\ \text{“)”}\ chtype\ \textbf{“between”}\ on\ \text{“.”}\ pn\ \textbf{“and”}\ on'\ \text{“.”}\ pn' \\
var & ::= & type\ vn\ [\text{“}=\text{”}\ ce] \\
type & ::= & \textbf{“Boolean”}\ \mid\ \textbf{“Integer”}\ \mid\ \textbf{“String”} \\
chtype & ::= & \textbf{“sync”}\ \mid\ \textbf{“async lossless”}\ \mid\ \textbf{“async lossy”},
\end{array}
$$

where the structure of model names $mn \in MN$, class names $cn \in CN$, object names $on \in ON$, channel names $chn \in CHN$, port names $pn \in PN$, variable names $vn \in VN$, and constant expressions $ce \in CE$ is left unspecified. We use a standard syntax for these concepts.

The syntax of state machines $sm \in \mathit{StateMachines}$, transitions $\mathit{trans} \in \mathit{Transitions}$, signal sending statements $\mathit{send} \in \mathit{Statements}$, signal reception statements $\mathit{rec} \in \mathit{Statements}$, and assignment

statements $assign \in Statements$ is defined as follows:

$$\begin{array}{lcl} sm & ::= & smn\ var^*\ states\ trans^* \\ states & ::= & \text{“}\textbf{initial}\text{”}\ sn\ sn^*\ [\ \text{“}\textbf{final}\text{”}\ sn^+\ ] \\ trans & ::= & tn\ \text{“}\textbf{from}\text{”}\ sn\ \text{“}\textbf{to}\text{”}\ sn'\ [\ send \mid rec \mid assign \mid delay \mid e\ ] \\ send & ::= & \text{“}\textbf{send}\text{”}\ sgn\ \text{“(”}\ e^*\ \text{“)”}\ \text{“}\textbf{to}\text{”}\ pn \\ rec & ::= & \text{“}\textbf{receive}\text{”}\ sgn\ \text{“(”}\ vn^*\ \text{“|”}\ e\ \text{“)”}\ \text{“}\textbf{from}\text{”}\ pn \\ assign & ::= & vn\ \text{“:=”}\ e \\ delay & ::= & \text{“}\textbf{after}\text{”}\ ice\ \text{“}\textbf{ms}\text{”}, \end{array}$$

where the structure of state machine names $smn \in SMN$, state names $sn \in SN$, transition names $tn \in TN$, signal names $sgn \in SGN$, expressions $e \in Expressions$, and integer constant expressions $ice \in ICE$ is left unspecified. Again, we use a standard syntax for these concepts. Because we do not consider time and successful termination, the delay statement and the notion of final states are left out of this syntax definition.

## A.3 Semantics

We use a variant of structural operational semantics [26, 18] that relies heavily on valuation functions to define the semantics of SLCO. These valuation functions enable a compositional definition of the semantics. The potential behavior of an SLCO model is defined in terms of the potential behavior of the objects that constitute this model. In turn, the potential behavior of the objects is defined in terms of the potential behavior of their classes, which is defined in terms of the potential behavior of the state machines that constitute the classes. Finally, the potential behavior of a state machine is defined in terms of the potential behavior of the transitions of the state machine.

### A.3.1 Transitions

The potential behavior of a transition is defined by the relation

$$\rightarrow_{TRANS} \subseteq (Transitions \times SN \times V_{VARS} \times V_{VARS}) \times TL \times (SN \times V_{VARS} \times V_{VARS}),$$

where each function $v_{vars}$ from the set of partial functions $V_{VARS} = VN \nrightarrow CE$ maps variable names to constant expressions, and the syntax of transition labels $l \in TL$ is defined as follows:

$$\begin{array}{lcl} l & ::= & \epsilon \\ & \mid & \text{“}\textbf{send}\text{”}\ sgn\ \text{“(”}\ ce^*\ \text{“)”}\ \text{“}\textbf{to}\text{”}\ pn \\ & \mid & \text{“}\textbf{receive}\text{”}\ sgn\ \text{“(”}\ ce^*\ \text{“)”}\ \text{“}\textbf{from}\text{”}\ pn \\ & \mid & \text{“}\textbf{send}\text{”}\ sgn\ \text{“(”}\ ce^*\ \text{“)”} \\ & \mid & \text{“}\textbf{receive}\text{”}\ sgn\ \text{“(”}\ ce^*\ \text{“)”} \\ & \mid & \text{“}\textbf{lost}\text{”}\ sgn\ \text{“(”}\ ce^*\ \text{“)”} \\ & \mid & sgn\ \text{“(”}\ ce^*\ \text{“)”} \\ & \mid & vn\ \text{“:=”}\ ce. \end{array}$$

The relation $\rightarrow_{TRANS}$ is the least relation satisfying the following rules:

$$\langle tn\ \textbf{from}\ sn\ \textbf{to}\ sn', sn, v_{vars}, v'_{vars}\rangle \xrightarrow{\epsilon}_{TRANS} \langle sn', v_{vars}, v'_{vars}\rangle \tag{T1}$$

$$\frac{\langle e, v_{vars}, v'_{vars}\rangle \Rightarrow_{EXP} \textbf{true}}{\langle tn\ \textbf{from}\ sn\ \textbf{to}\ sn'\ e, sn, v_{vars}, v'_{vars}\rangle \xrightarrow{\epsilon}_{TRANS} \langle sn', v_{vars}, v'_{vars}\rangle} \tag{T2}$$

$$\frac{\langle assign, v_{vars}, v'_{vars}\rangle \overset{l}{\Rightarrow}_{ASSIGN} \langle v''_{vars}, v'''_{vars}\rangle}{\langle tn \textbf{ from } sn \textbf{ to } sn' \ assign, sn, v_{vars}, v'_{vars}\rangle \xrightarrow{l}_{TRANS} \langle sn', v''_{vars}, v'''_{vars}\rangle} \quad \text{(T3)}$$

$$\frac{\langle e^*, v_{vars}, v'_{vars}\rangle \Rightarrow_{EXPS} ce^*}{\langle tn \textbf{ from } sn \textbf{ to } sn' \textbf{ send } sign(e^*) \textbf{ to } pn, sn, v_{vars}, v'_{vars}\rangle \xrightarrow{\textbf{send } sgn(ce^*) \textbf{ to } pn}_{TRANS} \langle sn', v_{vars}, v'_{vars}\rangle} \quad \text{(T4)}$$

$$\frac{\langle ce^*, vn^*, v_{vars}, v'_{vars}\rangle \Rightarrow_{SUB} \langle v''_{vars}, v'''_{vars}\rangle, \quad \langle e, v''_{vars}, v'''_{vars}\rangle \Rightarrow_{EXP} \textbf{true}}{\langle tn \textbf{ from } sn \textbf{ to } sn' \textbf{ receive } sgn(vn^* \mid e) \textbf{ from } pn, sn, v_{vars}, v'_{vars}\rangle \xrightarrow{\textbf{receive } sgn(ce^*) \textbf{ from } pn}_{TRANS} \langle sn', v''_{vars}, v'''_{vars}\rangle}. \quad \text{(T5)}$$

Rule (T1) defines that a transition specification *tn* **from** *sn* **to** *sn′* leads to a transition from state *sn* to state *sn′*, leaving the valuation functions $v_{vars}$ and $v'_{vars}$ unchanged.

Rule (T2) defines that such a transition is also possible given a transition specification *tn* **from** *sn* **to** *sn′* *e*, provided that the expression *e* evaluates to **true**. This rule refers to a relation $\Rightarrow_{EXP} \subseteq (Expressions \times V_{VARS} \times V_{VARS}) \times CE$, which defines how expressions $e \in Expressions$ evaluate to constant expressions $ce \in CE$, given two valuation functions $v_{vars} \in V_{VARS}$ and $v'_{vars} \in V_{VARS}$. We do not specify the syntax of expressions, as mentioned above, and leave their semantics unspecified as well. We use a standard semantics for the evaluation of expressions, where two valuation functions are used to distinguish between the local variables of state machines and the global variables of objects. This distinction is discussed in further detail below.

Rule (T3) defines that the execution of an assignment statement as part of a transition leads to an update of the valuation functions $v_{vars}$ and $v'_{vars}$. The rule refers to the relation $\Rightarrow_{ASSIGN} \subseteq (Expressions \times V_{VARS} \times V_{VARS}) \times TL \times (V_{VARS} \times V_{VARS})$, which defines the details of this update. The relation $\Rightarrow_{ASSIGN}$ is the least relation satisfying the following rules:

$$\frac{\langle e, v_{vars}, v'_{vars}\rangle \Rightarrow_{EXP} ce, \quad vn \in \mathrm{dom}(v'_{vars}), \quad v''_{vars} = v'_{vars}[ce/vn]}{\langle vn := e, v_{vars}, v'_{vars}\rangle \overset{vn := ce}{\Longrightarrow}_{ASSIGN} \langle v_{vars}, v''_{vars}\rangle}$$

$$\frac{\langle e, v_{vars}, v'_{vars}\rangle \Rightarrow_{EXP} ce, \quad vn \notin \mathrm{dom}(v'_{vars}), \quad vn \in \mathrm{dom}(v_{vars}), \quad v''_{vars} = v_{vars}[ce/vn]}{\langle vn := e, v_{vars}, v'_{vars}\rangle \overset{vn := ce}{\Longrightarrow}_{ASSIGN} \langle v''_{vars}, v'_{vars}\rangle}.$$

In these rules, the distinction between local and global variables becomes apparent. In both rules, valuation function $v_{vars}$ maps the global variables to their values, and function $v'_{vars}$ maps the local variables to their values. If a local variable named *vn* exists, denoted by $vn \in \mathrm{dom}(v'_{vars})$, then the valuation function $v'_{vars}$ is updated by mapping *vn* to *ce*. Otherwise, if a global variable named *vn* exists, the valuation function $v_{vars}$ is updated. We use $f[v/x]$ to denote the updated function $f$, where $f[v/x](x) = v$ and $f[v/x](y) = f(y)$ for all $y \neq x$.

Rule (T4) defines the semantics of transitions with statements that send signals. It refers to the relation $\Rightarrow_{EXPS} \subseteq (SEQ(Expressions) \times V_{VARS} \times V_{VARS}) \times SEQ(CE)$, which defines how a sequence of expressions is evaluated to a sequence of constant expressions, given two valuation functions. The relation $\Rightarrow_{EXPS}$ is the least relation that satisfies the following rules:

$$\langle \epsilon, v_{vars}, v'_{vars}\rangle \Rightarrow_{EXPS} \epsilon$$

$$\frac{\langle e, v_{vars}, v'_{vars}\rangle \Rightarrow_{EXP} ce, \quad \langle e^*, v_{vars}, v'_{vars}\rangle \Rightarrow_{EXPS} ce^*}{\langle e\ e^*, v_{vars}, v'_{vars}\rangle \Rightarrow_{EXPS} ce\ ce^*}.$$

Finally, an instance of rule (T5) exists for all $ce^* \in SEQ(CE)$. These instances define the semantics of transitions with signal reception statements. It refers to the relation $\Rightarrow_{SUB} \subseteq$

$(CE \times VN \times V_{VARS} \times V_{VARS}) \times (V_{VARS} \times V_{VARS})$, which defines sequential updates of the values of a set of variables. The relation $\Rightarrow_{SUB}$ is the least relation satisfying the following rules:

$$\langle \epsilon, \epsilon, v_{vars}, v'_{vars} \rangle \Rightarrow_{SUB} \langle v_{vars}, v'_{vars} \rangle$$

$$\frac{vn \in \mathrm{dom}(v'_{vars}), \quad v''_{vars} = v'_{vars}[ce/vn], \quad \langle ce^*, vn^*, v_{vars}, v''_{vars} \rangle \Rightarrow_{SUB} \langle v'''_{vars}, v''''_{vars} \rangle}{\langle ce\ ce^*, vn\ vn^*, v_{vars}, v'_{vars} \rangle \Rightarrow_{SUB} \langle v'''_{vars}, v''''_{vars} \rangle}$$

$$\frac{\begin{array}{c} vn \notin \mathrm{dom}(v'_{vars}), \quad vn \in \mathrm{dom}(v_{vars}), \quad v''_{vars} = v_{vars}[ce/vn], \\ \langle ce^*, vn^*, v''_{vars}, v'_{vars} \rangle \Rightarrow_{SUB} \langle v'''_{vars}, v''''_{vars} \rangle \end{array}}{\langle ce\ ce^*, vn\ vn^*, v_{vars}, v'_{vars} \rangle \Rightarrow_{SUB} \langle v'''_{vars}, v''''_{vars} \rangle}.$$

Each instance of rule (T5) specifies that a statement **receive** $sgn(vn^* \mid e)$ **from** $pn$ is only enabled if expression $e$ evaluates to **true** after updating the values of the variables $vn^*$ to the constant expressions $ce^*$. This sequence of constant expressions represents the possible values of the arguments of signals sent by other objects.

### A.3.2 State Machines

The potential behavior of a state machine is defined by the relation

$$\rightarrow_{SM} \subseteq (StateMachines \times S_{SMS} \times V_{VARS} \times V_{SMS}) \times TL \times (S_{SMS} \times V_{VARS} \times V_{SMS}),$$

where each function $s_{sms}$ from the set of partial functions $S_{SMS} = SMN \nrightarrow SN$ maps state machine names to state names, and each function $v_{sms}$ from the set of partial functions $V_{SMS} = SMN \nrightarrow (VN \nrightarrow CE)$ maps state machine names to functions that map variable names to constant expression. The fact that state machine $smn$ is in state $sn$ is encoded as $s_{sms}(smn) = sn$ using a function $s_{sms} \in S_{SMS}$. Furthermore, the fact that variable $vn$ of state machine $smn$ has the value $ce$ is encoded as $v_{sms}(smn)(vn) = ce$ using a function $v_{sms} \in V_{SMS}$. The relation $\rightarrow_{SM}$ is the least relation satisfying the following rule:

$$\frac{\begin{array}{c} trans \in trans^*, \\ \langle trans, s_{sms}(smn), v_{vars}, v_{sms}(smn) \rangle \xrightarrow{l}_{TRANS} \langle sn, v'_{vars}, v''_{vars} \rangle, \\ s'_{sms} = s_{sms}[sn/smn], \quad v'_{sms} = v_{sms}[v''_{vars}/smn] \end{array}}{\langle smn\ var^*\ states\ trans^*, s_{sms}, v_{vars}, v_{sms} \rangle \xrightarrow{l}_{SM} \langle s'_{sms}, v'_{vars}, v'_{sms} \rangle}. \tag{SM}$$

We use $e \in e^*$ to denote $\exists e', e'' \,.\, e^* \equiv e'\ e\ e''$, for each element $e$ of a syntactic category.

Rule (SM) defines that if one of the transitions of a state machine can go from state $s_{sms}(smn)$ to state $sn$ while performing an action represented by $l$, then this state machine can make a transition to the same state from state $s_{sms}(smn)$ while performing the same action.

### A.3.3 Classes

The potential behavior of a class is defined by the relation

$$\rightarrow_{CLASS} \subseteq (Classes \times S_{SMS} \times V_{VARS} \times V_{SMS}) \times TL \times (S_{SMS} \times V_{VARS} \times V_{SMS}).$$

The relation $\rightarrow_{CLASS}$ is the least relation satisfying the following rule:

$$\frac{sm \in sm^*, \quad \langle sm, s_{sms}, v_{vars}, v_{sms} \rangle \xrightarrow{l}_{SM} \langle s'_{sms}, v'_{vars}, v'_{sms} \rangle}{\langle cn\ var^*\ port^*\ sm^*, s_{sms}, v_{vars}, v_{sms} \rangle \xrightarrow{l}_{CLASS} \langle s'_{sms}, v'_{vars}, v'_{sms} \rangle}. \tag{C}$$

Rule (C) defines that the potential behavior of a class is derived from the potential behavior of the state machines of that class.

### A.3.4 Objects

The potential behavior of a set of objects is defined by the relation

$$\begin{aligned}\rightarrow_{OBJS} \subseteq (&\mathit{Objects} \times \mathit{Classes} \times \mathit{Channels} \times S_{OBJS} \times V_{GLOB} \times V_{LOC} \times B)\\ &\times TL \times (S_{OBJS} \times V_{GLOB} \times V_{LOC} \times B),\end{aligned}$$

where each function $v_{glob}$ from the set of partial functions $V_{GLOB} = ON \nrightarrow (VN \nrightarrow CE)$ maps object names to functions that map variable names to constant expressions, each function $v_{loc}$ from the set of partial functions $V_{LOC} = ON \nrightarrow (SMN \nrightarrow (VN \nrightarrow CE))$ maps object names to functions that map state machine names to functions that map variable names to constant expressions, each function $s_{objs}$ from the set of partial functions $S_{OBJS} = ON \nrightarrow (SMN \nrightarrow SN)$ maps object names to functions that map state machine names to state names, and each function $b$ from the set of partial functions $B = (CHN \times ON \times ON) \nrightarrow (SGN \times SEQ(CE)) \cup \{\mathbf{nil}\}$ maps tuples consisting of a channel name and two object names to the constant nil or a tuple consisting of a signal name and a sequence of constant expressions. The functions in $V_{GLOB}$ encode valuations of global variables, and the functions in $V_{LOC}$ encode valuations of local variables. The set of functions $S_{OBJS}$ is an extension of the set $S_{SMS}$. The fact that state machine $smn$ of object $on$ is in state $sn$ is encoded as $s_{objs}(on)(smn) = sn$ using a function $s_{objs} \in S_{OBJS}$. The functions in $B$ are used to encode the content of a set of buffers. The fact that the buffer corresponding to the channel $chn$ that connects objects $on_1$ and $on_2$ is empty is encoded as $b(chn, on_1, on_2) = \mathbf{nil}$ using a function $b \in B$. In the remainder of this section, we discuss a number of the rules that define this relation.

The following rule defines the part of the semantics of sequences of objects related to assignment statements:

$$\frac{\begin{array}{c}on : cn \in obj^*, \quad cn\ var^*\ pn^*\ sm^* \in class^*,\\ \langle cn\ var^*\ pn^*\ sm^*, s_{objs}(on), v_{glob}(on), v_{loc}(on)\rangle\\ \xrightarrow{vn := ce}_{CLASS} \langle s_{sms}, v_{vars}, v_{sms}\rangle,\\ s'_{objs} = s_{objs}[s_{sms}/on], \quad v'_{glob} = v_{glob}[v_{vars}/on], \quad v'_{loc} = v_{loc}[v_{sms}/on]\end{array}}{\langle obj^*, class^*, chan^*, s_{objs}, v_{glob}, v_{loc}, b\rangle \xrightarrow{vn := ce}_{OBJS} \langle s'_{objs}, v'_{glob}, v'_{loc}, b\rangle}. \tag{O1}$$

Rule (O1) specifies that a sequence of objects can perform an assignment $vn := ce$ if one of the objects in the sequence is an instance of a class that can perform this assignment.

The following rules defines the part of the semantics of sequences of objects related to synchronous communication:

$$\frac{\begin{array}{c}on_1 : cn_1 \in obj^*, \quad on_2 : cn_2 \in obj^*,\\ cn_1\ var_1^*\ pn_1^*\ sm_1^* \in class^*, \quad cn_2\ var_2^*\ pn_2^*\ sm_2^* \in class^*,\\ chn(type^*)\ \mathbf{sync\ from}\ on_1.pn_1\ \mathbf{to}\ on_2.pn_2 \in chan^*,\\ \langle cn_1\ var_1^*\ pn_1^*\ sm_1^*, s_{objs}(on_1), v_{glob}(on_1), v_{loc}(on_1)\rangle\\ \xrightarrow{\mathbf{send}\ sgn(ce^*)\ \mathbf{to}\ pn_1}_{CLASS} \langle s_{sms}, v_{vars}, v_{sms}\rangle,\\ \langle cn_2\ var_2^*\ pn_2^*\ sm_2^*, s_{objs}(on_2), v_{glob}(on_2), v_{loc}(on_2)\rangle\\ \xrightarrow{\mathbf{receive}\ sgn(ce^*)\ \mathbf{from}\ pn_2}_{CLASS} \langle s'_{sms}, v'_{vars}, v'_{sms}\rangle,\\ s'_{objs} = s_{objs}[s_{sms}/on_1][s'_{sms}/on_2],\\ v'_{glob} = v_{glob}[v'_{vars}/on_2], \quad v'_{loc} = v_{loc}[v'_{sms}/on_2]\end{array}}{\langle obj^*, class^*, chan^*, s_{objs}, v_{glob}, v_{loc}, b\rangle \xrightarrow{sn(ce^*)}_{OBJS} \langle s'_{objs}, v'_{glob}, v'_{loc}, b\rangle} \tag{O2}$$

$$\frac{\begin{array}{c} on_1 : cn_1 \in obj^*, \quad on_2 : cn_2 \in obj^*, \\ cn_1\ var_1^*\ pn_1^*\ sm_1^* \in class^*, \quad cn_2\ var_2^*\ pn_2^*\ sm_2^* \in class^*, \\ chn(type^*)\ \textbf{sync between}\ on_1.pn_1\ \textbf{and}\ on_2.pn_2 \in chan^*, \\ \langle cn_1\ var_1^*\ pn_1^*\ sm_1^*, s_{objs}(on_1), v_{glob}(on_1), v_{loc}(on_1)\rangle \\ \xrightarrow{\textbf{send}\ sgn(ce^*)\ \textbf{to}\ pn_1}_{CLASS} \langle s_{sms}, v_{vars}, v_{sms}\rangle, \\ \langle cn_2\ var_2^*\ pn_2^*\ sm_2^*, s_{objs}(on_2), v_{glob}(on_2), v_{loc}(on_2)\rangle \\ \xrightarrow{\textbf{receive}\ sgn(ce^*)\ \textbf{from}\ pn_2}_{CLASS} \langle s'_{sms}, v'_{vars}, v'_{sms}\rangle, \\ s'_{objs} = s_{objs}[s_{sms}/on_1][s'_{sms}/on_2], \\ v'_{glob} = v_{glob}[v'_{vars}/on_2], \quad v'_{loc} = v_{loc}[v'_{sms}/on_2] \end{array}}{\langle obj^*, class^*, chan^*, s_{objs}, v_{glob}, v_{loc}, b\rangle \xrightarrow{sn(ce^*)}_{OBJS} \langle s'_{objs}, v'_{glob}, v'_{loc}, b\rangle} \tag{O3}$$

$$\frac{\begin{array}{c} on_1 : cn_1 \in obj^*, \quad on_2 : cn_2 \in obj^*, \\ cn_1\ var_1^*\ pn_1^*\ sm_1^* \in class^*, \quad cn_2\ var_2^*\ pn_2^*\ sm_2^* \in class^*, \\ chn(type^*)\ \textbf{sync between}\ on_2.pn_2\ \textbf{and}\ on_1.pn_1 \in chan^*, \\ \langle cn_1\ var_1^*\ pn_1^*\ sm_1^*, s_{objs}(on_1), v_{glob}(on_1), v_{loc}(on_1)\rangle \\ \xrightarrow{\textbf{send}\ sgn(ce^*)\ \textbf{to}\ pn_1}_{CLASS} \langle s_{sms}, v_{vars}, v_{sms}\rangle, \\ \langle cn_2\ var_2^*\ pn_2^*\ sm_2^*, s_{objs}(on_2), v_{glob}(on_2), v_{loc}(on_2)\rangle \\ \xrightarrow{\textbf{receive}\ sgn(ce^*)\ \textbf{from}\ pn_2}_{CLASS} \langle s'_{sms}, v'_{vars}, v'_{sms}\rangle, \\ s'_{objs} = s_{objs}[s_{sms}/on_1][s'_{sms}/on_2], \\ v'_{glob} = v_{glob}[v'_{vars}/on_2], \quad v'_{loc} = v_{loc}[v'_{sms}/on_2] \end{array}}{\langle obj^*, class^*, chan^*, s_{objs}, v_{glob}, v_{loc}, b\rangle \xrightarrow{sn(ce^*)}_{OBJS} \langle s'_{objs}, v'_{glob}, v'_{loc}, b\rangle}. \tag{O4}$$

Rule (O2) specifies synchronous communication between two objects over a unidirectional channel, and Rules (O3) and (O4) specify synchronous communication over bidirectional channels. Because sending a signal does not affect the valuation of variables, the updates of $v_{glob}$ and $v_{loc}$ do not take object $on_1$ into account.

The following rules define asynchronous communication over a lossless channel:

$$\frac{\begin{array}{c} on_1 : cn_1 \in obj^*, \quad cn_1\ var_1^*\ pn_1^*\ sm_1^* \in class^*, \\ chn(type^*)\ \textbf{async lossless from}\ on_1.pn_1\ \textbf{to}\ on_2.pn_2 \in chan^*, \\ \langle cn_1\ var_1^*\ pn_1^*\ sm_1^*, s_{objs}(on_1), v_{glob}(on_1), v_{loc}(on_1)\rangle \\ \xrightarrow{\textbf{send}\ sgn(ce^*)\ \textbf{to}\ pn_1}_{CLASS} \langle s_{sms}, v_{vars}, v_{sms}\rangle, \\ s'_{objs} = s_{objs}[s_{sms}/on_1], \\ b(\langle chn, on_1, on_2\rangle) = \textbf{nil}, \quad b' = b[\langle sgn, ce^*\rangle/\langle chn, on_1, on_2\rangle] \end{array}}{\langle obj^*, class^*, chan^*, s_{objs}, v_{glob}, v_{loc}, b\rangle \xrightarrow{\textbf{send}\ sgn(ce^*)}_{OBJS} \langle s'_{objs}, v_{glob}, v_{loc}, b'\rangle} \tag{O5}$$

$$\frac{\begin{array}{c} on_2 : cn_2 \in obj^*, \quad cn_2\ var_2^*\ pn_2^*\ sm_2^* \in class^*, \\ chn(type^*)\ \textbf{async lossless from}\ on_1.pn_1\ \textbf{to}\ on_2.pn_2 \in chan^*, \\ \langle cn_2\ var_2^*\ pn_2^*\ sm_2^*, s_{objs}(on_2), v_{glob}(on_2), v_{loc}(on_2)\rangle \\ \xrightarrow{\textbf{receive}\ sgn(ce^*)\ \textbf{from}\ pn_2}_{CLASS} \langle s_{sms}, v_{vars}, v_{sms}\rangle, \\ s'_{objs} = s_{objs}[s_{sms}/on_2], \\ v'_{glob} = v_{glob}[v_{vars}/on_2], \quad v'_{loc} = v_{loc}[v_{sms}/on_2], \\ b(\langle chn, on_1, on_2\rangle) = \langle sgn, ce^*\rangle, \quad b' = b[\textbf{nil}/\langle chn, on_1, on_2\rangle] \end{array}}{\begin{array}{c}\langle obj^*, class^*, chan^*, s_{objs}, v_{glob}, v_{loc}, b\rangle \\ \xrightarrow{\textbf{receive}\ sgn(ce^*)}_{OBJS} \langle s'_{objs}, v'_{glob}, v'_{loc}, b'\rangle\end{array}}. \tag{O6}$$

We only give the rules related to unidirectional channels. The rules for bidirectional channels are similar, however, and can be derived from the rules given above.

Rule (O5) specifies that a signal is placed in the buffer that corresponds to an asynchronous channel if an object sends this signal over the channel. This is only possible if the buffer is empty when the statement is executed. Rule (O6) specifies that a signal is removed from the buffer that corresponds to an asynchronous channel if an object that is connected to this channel is able to receive this signal.

The following rules define how asynchronous communication over a lossy channel differs from asynchronous communication over a lossless channel:

$$\frac{\begin{array}{c} on_1 : cn_1 \in obj^*, \quad cn_1\ var_1^*\ pn_1^*\ sm_1^* \in class^*, \\ chn(type^*)\ \textbf{async lossy from}\ on_1.pn_1\ \textbf{to}\ on_2.pn_2 \in chan^*, \\ \langle cn_1\ var_1^*\ pn_1^*\ sm_1^*, s_{objs}(on_1), v_{glob}(on_1), v_{loc}(on_1)\rangle \\ \xrightarrow{\textbf{send}\ sgn(ce^*)\ \textbf{to}\ pn_1}_{CLASS} \langle s_{sms}, v_{vars}, v_{sms}\rangle, \\ s'_{objs} = s_{objs}[s_{sms}/on_1] \end{array}}{\langle obj^*, class^*, chan^*, s_{objs}, v_{glob}, v_{loc}, B\rangle \xrightarrow{\textbf{lost}\ sgn(ce^*)}_{OBJS} \langle s'_{objs}, v'_{glob}, v'_{loc}, B\rangle} \tag{O7}$$

$$\frac{\begin{array}{c} on_1 : cn_1 \in obj^*, \quad cn_1\ var_1^*\ pn_1^*\ sm_1^* \in class^*, \\ chn(type^*)\ \textbf{async lossy from}\ on_1.pn_1\ \textbf{to}\ on_2.pn_2 \in chan^*, \\ \langle cn_1\ var_1^*\ pn_1^*\ sm_1^*, s_{objs}(on_1), v_{glob}(on_1), v_{loc}(on_1)\rangle \\ \xrightarrow{\textbf{send}\ sgn(ce^*)\ \textbf{to}\ pn_1}_{CLASS} \langle s_{sms}, v_{vars}, v_{sms}\rangle, \\ s'_{objs} = s_{objs}[s_{sms}/on_1], \\ b(\langle chn, on_1, on_2\rangle) = \langle sgn', ce^{*\prime}\rangle, \quad b' = b[\langle sgn, ce^*\rangle/\langle chn, on_1, on_2\rangle] \end{array}}{\langle obj^*, class^*, chan^*, s_{objs}, v_{glob}, v_{loc}, b\rangle \xrightarrow{\textbf{lost}\ sgn'(ce^{*\prime})}_{OBJS} \langle s'_{objs}, v'_{glob}, v'_{loc}, b'\rangle}. \tag{O8}$$

Besides the rules shown above, additional rules exist that complete the definition of asynchronous communication over lossy channels. These rules are similar to the rules defining communication over asynchronous, lossless channels and can be derived from those rules.

Rule (O7) specifies that a signal sent over a lossy channel may get lost. In this rule, the function representing the buffer is unchanged. Rule (O8) specifies an alternative way of losing signals. It shows that a signal sent over a lossy channel can be placed in the corresponding buffer, even if this buffer already contains a signal. The new signal replaces the existing signal, which means that the original signal is lost.

### A.3.5 Models

Finally, the potential behavior of a model is defined by the relation

$$\begin{aligned} \rightarrow_{MODEL} \subseteq\ & (Models \times S_{OBJS} \times V_{GLOB} \times V_{LOC} \times B) \\ & \times TL \times (Models \times S_{OBJS} \times V_{GLOB} \times V_{LOC} \times B), \end{aligned}$$

which is the least relation satisfying the following rule:

$$\frac{\begin{array}{c} m \equiv mn\ obj^*\ class^*\ chan^*, \\ \langle obj^*, class^*, chan^*, s_{objs}, v_{glob}, v_{loc}, b\rangle \xrightarrow{l}_{OBJS} \langle s'_{objs}, v'_{glob}, v'_{loc}, b'\rangle \end{array}}{\langle m, s_{objs}, v_{glob}, v_{loc}, b\rangle \xrightarrow{l}_{MODEL} \langle m, s'_{objs}, v'_{glob}, v'_{loc}, b'\rangle}. \tag{M}$$

## A.4 Initialization

By specifying which configurations $\langle m, s_{objs}, v_{glob}, v_{loc}, b\rangle$ and $\langle m, s'_{objs}, v'_{glob}, v'_{loc}, b'\rangle$ are related, rule (M) defines the steps that can be taken according to model $m$. The initial configuration is defined by choosing appropriate functions $s_{objs}$, $v_{glob}$, $v_{loc}$, and $b$. Below, we define a number of functions that map SLCO models to the functions that define the initial configurations of these models.

### A.4.1 Initial States

The function $S^M_{OBJS} : \mathit{Models} \to (\mathit{ON} \nrightarrow (\mathit{SMN} \nrightarrow \mathit{SN}))$ is defined as follows:

$$S^M_{OBJS}(mn\ class^*\ obj^*\ chan^*) = \\ \{(on, S^C_{SMS}(cn\ var^*\ pn^*\ sm^*)) \mid cn\ var^*\ pn^*\ sm^* \in class^* \wedge on : cn \in obj^*\}.$$

It maps models to functions from $S_{OBJS}$, such that each name of a state machine is mapped to its initial state as defined by the model. The definition refers to the function $S^C_{SMS} : \mathit{Classes} \to (\mathit{SMN} \nrightarrow \mathit{SN})$, which is defined as follows:

$$S^C_{SMS}(cn\ var^*\ pn^*\ sm^*) = \\ \{(smn, sn) \mid smn\ \textbf{initial}\ sn\ sn^* \in sm^*\} \cup \\ \{(smn, sn) \mid smn\ \textbf{initial}\ sn\ sn^*\ \textbf{final}\ sn^+ \in sm^*\}.$$

This function maps classes to functions from $S_{SMS}$, such that each name of a state machine is mapped to its initial state as defined by the class.

### A.4.2 Initial Values of Variables

The function $V^M_{GLOB} : \mathit{Models} \to (\mathit{ON} \nrightarrow (\mathit{VN} \nrightarrow \mathit{CE}))$ is defined as follows:

$$V^M_{GLOB}(mn\ class^*\ obj^*\ chan^*) = \\ \{(on, V^V(var^*)) \mid cn\ var^*\ pn^*\ sm^* \in class^* \wedge on : cn \in obj^*\}.$$

It maps models to functions from $V_{GLOB}$, such that each global variable is mapped to its initial value as specified by the model. The definition refers to a function $V^V : \mathit{SEQ}(\mathit{Variables}) \to (\mathit{VN} \nrightarrow \mathit{CE})$, which is defined as follows:

$$V^V(var^*) = \\ \{(vn, \textbf{false}) \mid \textbf{Boolean}\ vn \in var^*\} \cup \{(vn, bc) \mid \textbf{Boolean}\ vn = bc \in var^*\} \cup \\ \{(vn, 0) \mid \textbf{Integer}\ vn \in var^*\} \cup \{(vn, ic) \mid \textbf{Integer}\ vn = ic \in var^*\} \cup \\ \{(vn, \text{“”}) \mid \textbf{String}\ vn \in var^*\} \cup \{(vn, sc) \mid \textbf{String}\ vn = sc \in var^*\}.$$

This function maps sequences of variable declarations to functions that map variable names to the appropriate initial values as specified by the sequence of declarations.

Similar functions $V^M_{LOC} : \mathit{Models} \to (\mathit{ON} \nrightarrow (\mathit{SMN} \nrightarrow (\mathit{VN} \nrightarrow \mathit{CE})))$ and $V^C : \mathit{Classes} \to (\mathit{SMN} \nrightarrow (\mathit{VN} \nrightarrow \mathit{CE}))$ exist that are related to the initial values of local variables. These functions are defined as follows:

$$V^M_{LOC}(mn\ class^*\ obj^*\ chan^*) = \\ \{(on, V^C(cn\ var^*\ pn^*\ sm^*)) \mid cn\ var^*\ pn^*\ sm^* \in class^* \wedge on : cn \in obj^*\}$$

$$V^C(cn\ var^*\ pn^*\ sm^*) = \{(smn, V^V(var^*) \mid smn\ var^*\ states\ trans^* \in sm^*\}.$$

### A.4.3 Buffers

The function $B^M : \mathit{Models} \to ((\mathit{CHN} \times \mathit{ON} \times \mathit{ON}) \nrightarrow (\mathit{SGN} \times \mathit{SEQ}(\mathit{CE})) \cup \{\mathbf{nil}\})$ is defined as follows:

$$
\begin{array}{l}
B^M(mn\ class^*\ obj^*\ chan^*) = \{(\langle chn, on_1, on_2\rangle, \mathbf{nil}) \mid \\
\quad chn(type^*)\ \mathbf{async\ lossless\ from}\ on_1.pn_1\ \mathbf{to}\ on_2.pn_2 \in chan^* \vee \\
\quad chn(type^*)\ \mathbf{async\ lossless\ between}\ on_1.pn_1\ \mathbf{and}\ on_2.pn_2 \in chan^* \vee \\
\quad chn(type^*)\ \mathbf{async\ lossless\ between}\ on_2.pn_2\ \mathbf{and}\ on_1.pn_1 \in chan^* \vee \\
\quad chn(type^*)\ \mathbf{async\ lossy\ from}\ on_1.pn_1\ \mathbf{to}\ on_2.pn_2 \in chan^* \vee \\
\quad chn(type^*)\ \mathbf{async\ lossy\ between}\ on_1.pn_1\ \mathbf{and}\ on_2.pn_2 \in chan^* \vee \\
\quad chn(type^*)\ \mathbf{async\ lossy\ between}\ on_2.pn_2\ \mathbf{and}\ on_1.pn_1 \in chan^* \\
\}.
\end{array}
$$

It maps models to functions from $B$, such that each buffer corresponding to a channel in the model is empty.