Eindhoven University of Technology

MASTER

Investigating the usefulness of Domain-Specific Transformation Languages

de Graaf, M.C.J.

Award date:
2016

Link to publication

Disclaimer
This document contains a student thesis (bachelor's or master's), as authored by a student at Eindhoven University of Technology. Student theses are made available in the TU/e repository upon obtaining the required degree. The grade received is not published on the document as presented in the repository. The required complexity or quality of research of student theses may vary by program, and the required minimum study period may vary in duration.

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain

Investigating the usefulness of Domain-Specific Transformation Languages

Master thesis of *M.C.J. (Marijn) de Graaf BSc* at Eindhoven University of Technology. May, 2016.

Keywords: domain-specific transformation language, model transformation language, domain-specific language, model driven engineering, SLCO.

Abstract

Model-Driven Engineering (MDE) is a software development methodology based around *models* and *model transformations*. In the field of MDE, model transformations are generally specified using *General-Purpose Transformation Languages* (GPTLs). *Domain experts* can write the models, in *Domain-Specific Languages* (DSLs). But they cannot write the model transformations, at least not without knowing the *abstract syntax* of the DSL. This is because these languages describe the transformations in terms of the abstract syntax whereas the domain experts often only know the *concrete syntax*. Therefore, some researchers suggest the use of *Domain-Specific Transformation Languages* (DSTLs). They allow domain experts to participate in the development of model transformations by enabling them to use the concrete syntax to specify the transformations. In other words, the concrete syntax of the source- and/or target DSL is contained as part of the DSTL.

Though DSTLs have been applied to simple cases, it has not yet been researched whether they are also useful for more complicated cases, that are common in practice. Therefore, *we have investigated how useful it is to employ DSTLs in practice*. To do this we performed a case study. We created a DSTL for a DSL called SLCO. We implemented several transformations in this DSTL, and compared them with a traditional implementation.

We have found that DSTLs are practical for relatively simple transformations, but that as the transformations become more complex, the advantages quickly diminish and it becomes easier to use more traditional paradigms based on the abstract syntax (i.e. traditional GPTLs) than DSTLs.

Department
Department of Mathematics
and Computer science

Section
Software Engineering
and Technology

Assessment committee

dr.ing. A.J. (Anton) Wijs	(Supervisor)
S.M.J. (Sander) de Putter MSc	(Supervisor)
dr. R. (Ruurd) Kuiper	(Internal)
dr.ir. T.A.C. (Tim) Willemse	(External)

Table of contents

1 Introduction

A lot of background and concepts have to be introduced before the goal and approach of our research can be explained. To give the readers, especially those already familiar with (some of) the concepts, an idea of the goal of this research as soon as possible, we first provide a ***summary*** *of the goal and approach of our research. The background and concepts used in this summary are explained in* **Section 1.1: 'Background and concepts'**.

In the field of *model-driven engineering*, *model transformations* are generally specified using *general-purpose transformation languages*. Models are often expressed in a *domain-specific language* (DSL). This can be done by domain experts. The model *transformations* however, can*not* be expressed by the domain experts, at least not without knowing the *abstract syntax* of the DSL. This is because the common model transformation languages specify transformations in terms of the abstract syntax of the source- and target languages, while the domain experts generally only know the *concrete syntax*.

To help domain experts to also specify model *transformations*, some researchers suggest the use of *Domain-Specific Transformation Languages* (DSTLs). In DSTLs, transformations can be expressed in terms of the concrete syntax.

Though DSTLs have been applied to simple cases, it has not yet been researched whether they are also useful for more complicated cases, that are common in practice. Therefore, we have investigated *how useful DSTLs are when used in practice*. We performed a case study. We created a DSTL for SLCO, a simple but nontrivial DSL that is advanced enough to be used for practical applications. We implemented several transformations in this DSTL, and compared them with a traditional implementation. This shows in which cases DSTLs are convenient, and in which cases their limitations are reached.

1.1 Background and concepts

1.1.1 Model-Driven Engineering

Model-Driven Engineering (**MDE**) [1] is a software development methodology based around **models** and **model transformations** (**Figure 1**) [2]. MDE raises the level of abstraction in program specification [3]. This enables higher reusability [4]. It also increases the ability of domain experts to participate in software development. A **domain expert** is someone who has high knowledge of the problem domain, but typically has little knowledge of aspects specific to writing software.

Engineers can express aspects of a system at the appropriate level of abstraction using models [2]. For example, there are the several kinds of diagrams in UML [5], in which models can specify various aspects of a system at various levels of detail, though models can also be specified in other ways than by using UML. Model transformations (and chains of them) transform the models into other models or other artifacts. A model can for example be transformed to be executable on a certain platform, or into a different model with certain desirable properties (for example by refactoring the model or to make it formally verifiable by a verification tool, such as SPIN [6, 7]).

1.1.2 Domain-Specific Languages

A model is specified in some model notation or modeling language. This modeling language is often a Domain-Specific Language (**DSL**) [4]. A DSL is a language tailored to a specific domain [8, 9]. A DSL can have a graphical (e.g. **Figure 2**, p. 14) or a textual notation (e.g. **Code fragment 1**, p. 13). Because DSLs are tailored to a specific domain, domain experts can use them to participate in the development of the system that is being designed.

Though DSLs allow domain experts to understand programs and models written in these DSLs, DSLs also have some drawbacks [9]. Each DSL (and toolset) takes time to create [9] and learn [9], while its applicability is limited to its problem domain [10]. That DSLs are specific to a domain leads to the existence of many DSLs. Each DSL needs to be learned by the person using it. Some researchers are opposed to a proliferation of DSLs [11]. They think that the benefits of a domain-specific notation does not outweigh the cost of creating and learning a DSL. So it is still under debate whether DSLs should be used[1].

DSLs (and languages in general) have a **concrete syntax** and an **abstract syntax**. The concrete syntax (e.g. **Code fragment 7**, pp. 26-27) is the format that the user has to adhere to while typing. What the user sees is for example **Code fragment 1** (p. 13). The abstract syntax (e.g. **Figure 3**, p. 14) is the structure of the language. It can (a.o.) be specified using a UML class diagram. The abstract syntax is also called **domain model** or **metamodel**.

HTML is an example of a DSL in the domain of web page markup, and SQL is a DSL in the domain of databases. There are also thousands of lesser known DSLs.

One of these languages is **SLCO** [12, 13] (e.g. **Code fragment 1** (p. 13), e.g. **Figure 2** (p. 14), **Figure 3** (p. 14)). SLCO is short for Simple Language of Communicating Objects. SLCO can specify systems consisting of objects that operate in parallel and communicate. The behavior of these objects is specified with state machines with extensions specific to SLCO. SLCO has a graphical and a textual notation. We have created a transformation language tailored to SLCO.

1.1.3 General-Purpose Transformation Languages

A model transformation is often expressed in a language specifically designed for model transformations (a **model transformation language**), although it could also be done in a general-purpose language (GPL) [14]. Examples of model transformation languages are ATL, QVT, and languages in the Epsilon family (specifically ETL).

Model transformation languages are also DSLs. They are DSLs in the domain of model transformations. But they are not specific to performing transformations *on* a certain domain [14]. Neither the input model(s) nor the output model(s) of the transformation have to be in a specific DSL. Commonly used model transformation languages that are not specific to performing transformations on a certain domain, languages such as ATL, are also called general-purpose transformation languages (**GPTL**s) [14].

1.1.4 Domain-Specific Transformation Languages

In MDE, domain experts can create the models in DSLs specific to the domain of those models. But they cannot write the model *transformations*, at least not without knowing the abstract syntax of the DSLs. That is why Domain-Specific Transformation Languages (**DSTL**s) [15] have been invented. DSTLs (e.g. **Code fragment 3**, p. 19) can use knowledge specific to the domain that the transformation is performed on. In DSTLs, domain experts *can use the concrete syntax* of the source- and target DSLs [4]. This enables them to write model transformations while using as much of the syntaxes that they already know as possible.

Additionally to the abstract syntax of the source- and target DSLs, the user of a GPTL also needs to know the generic part of the transformation language, the part not specifically related to the DSLs. This is the part needed for the transformation mechanism. Just like the abstract syntax of the DSLs,

[1] The same is true for domain-specific transformation languages, explained in **Section 1.1.4: 'Domain-Specific Transformation Languages'**.

domain experts often do not know this generic part either. DSTLs do not solve that problem, because for DSTLs the generic part is still needed, but in DSTLs at least domain experts can use the concrete syntax of the DSLs, which they know, instead of the abstract syntax, which they do not know. So they need to learn less.

There are two kinds of DSTLs [14]: DSTLs that are tailored to the domain of their source and target languages, and DSTLs that are tailored to a specific kind of transformation, for example refactoring, model merging, migration or aspect weaving. We have investigated the first kind.

Since the DSTLs reuse the concrete syntaxes of DSLs and DSLs are optimized for their domain, DSTLs inherit their *advantages*. DSTLs should be easier to read and write for domain experts [2, 4, 15], closer to the way it looks in the source and target and therefore easier to learn [15], and possibly more compact. DSTLs can also encapsulate domain knowledge that otherwise needs to be repeatedly embedded in transformations in the general-purpose transformation language [14].

DSTLs also inherit the *drawbacks* of DSLs. It is only useful to create a DSTL when it can be used for enough transformations. The benefits have to be enough to outweigh the cost of creating and learning the DSTL. Even though DSTLs can provide domain experts and developers with advantages because of their suitability to the domain, just like for DSLs, it is still under debate whether DSTLs should be used, because some researchers fear a proliferation of DS(T)Ls, which means a separate DS(T)L needs to be learned for each application domain. Additionally, we show in this paper that using the concrete syntax of a DSL in a transformation, has limitations to its convenience in practice.

1.1.5 Thesis Statement

Though DSTLs have been applied to simple cases, it has not yet been researched whether they are also useful for more complicated cases, that are common in practice. Therefore, we have investigated *whether the theoretical advantages of DSTLs hold up when they are used in practice*.

To do this we performed a case study. We created a DSTL for the DSL called SLCO. We used SLCO, because it is a simple but nontrivial language, that is advanced enough to be used for practical applications. For example, it can specify the behavior of a LEGO® Mindstorms® [16, 13] robot and be mapped to a verification language [13] (by a transformation).

We implemented several transformations in the DSTL we created, and compared them with a traditional implementation. This shows in which cases DSTLs are convenient, and in which cases their limitations are reached.

1.2 Overview

First, we will discuss relevant literature in **Section 2: 'Related work'**. Then, we will explain our methodology in **Section 3: 'Using a DSTL in practice'**. After that, we will provide a short introduction into **Section 4: 'SLCO'**, the DSL for which we created a DSTL. In **Section 5: 'Initial design'**, we will present a design for the transformation language. After that, we will explain how we implemented it, in **Section 6: 'Implementation'**. In **Section 7: 'The limitations of our DSTL'**, we discuss in what cases our DSTL is not convenient in practice. Next, we will present extensions and improvements to the design in **Section 8: 'Extending the design'**. Finally, we will present our conclusions in **Section 9: 'Conclusions'**, and present possibilities for future work in **Section 10: 'Future work'**.

Figures and code fragments can generally be found at the end of the section they most belong to. If a figures or code fragment is in a different section, the page number is added.

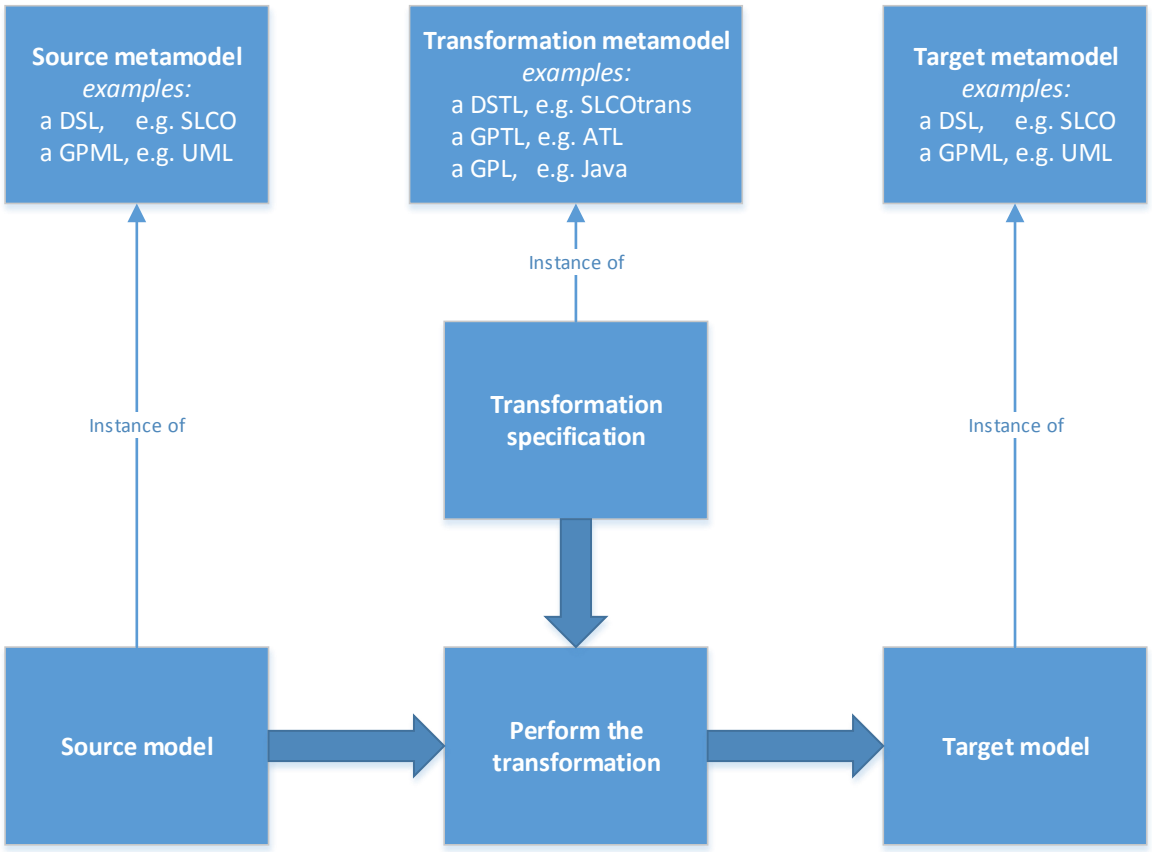

Figure 1: Performing a model transformation*. The transformation instructions in the transformation specification are applied to the source model which results in the target model. (GPML is short for General-Purpose Modeling Language).*

2 Related work

2.1 DSTLs

Bernhard Rumpe and Ingo Weisemöller *introduced the concept of DSTLs* [15]. They claim DSTLs are "more comprehensible and easier to learn for domain experts" than GPTLs. We have investigated whether it is also convenient to use DSTLs in practice.

In their paper, they "present a transformation language [(a DSTL)] that reuses the concrete syntax of a textual modeling language for hierarchical automata, which allows domain experts to describe models as well as modifications of models in a convenient, yet precise manner."

Rumpe and Weisemöller have demonstrated their DSTL on a transformation used in the process of flattening hierarchical automata. This is a simplified case of the transformations for flattening UML state machines. This is only a simple transformation though. In this paper, we also implement more complex transformations and investigate whether DSTLs have limitations that prevent more complicated transformations from being expressed conveniently.

2.2 Implementing DSTLs

Jerónimo Irazábal, Claudia Pons and Carlos Neil proposed to *implement DSTLs using a GPTL* [2]. At the same time, this is also used to define the semantics. We used their approach and implemented our DSTL using a GPTL too. They used ATL [17] in their demonstration however, whereas we used EOL [18, 19] (from the Epsilon family [20, 21, 22]).

In their paper, they "[first] present the main features of the proposal to define domain specific languages using transformation languages. [Then they] illustrate the use of the approach by the definition of a DSTL for the transformation of extended relational models. [After that, they] show relevant parts of the ATL-based implementation of such DSTL. [Finally, they] discuss an alternative implementation approach [(based on generating the ATL code instead of creating an interpreter) and they] compare this approach with related research."

It should be noted that Irazábal, Pons and Neil used a different kind of DSTLs than those we are investigating in this paper. Their example DSTL can combine transformations commonly used for extended relational models, similar to calling functions from a library. The DSTLs we are investigating are based on pattern matching, like those from the paper of Rumpe and Weisemöller. Their example DSTL is not tailored to the source- and target DSLs, but to the transformations common in the field of extended relational models. Despite this, the technique of using GPTLs to implement DSTLs that they introduced, could still be used by us.

Jesús Sánchez Cuadrado, Esther Guerra, and Juan de Lara proposed a different way to implement DSTLs [14]. *They created a DSL to describe DSTLs*. Their DSL describes how the DSTL is derived from its corresponding DSL. We did not use their DSL though, because we were already familiar with some GPTLs and the time span of our project did not allow us to investigate this technique thoroughly enough.

In their paper, they "propose a framework for the systematic creation of DSTLs. First, [they] look into the characteristics of domain-specific transformation tools, deriving a categorization which is the basis of [their] framework. Then, [they] propose a domain-specific language to describe DSTLs, from which [they] derive a ready-to-run workbench which includes the abstract syntax, concrete syntax and translational semantics of the DSTL."

2.3 SLCO

To investigate whether DSTLs are useful in practice, we created a DSTL tailored to SLCO. *SLCO* is described in the PhD Thesis of Luc Engelen [13] to investigate several ideas about MDE and DSLs, and to demonstrate the usefulness of DSLs in practice. To do this, Engelen investigated whether it is possible to develop the software of a conveyor belt created with LEGO® Mindstorms® [16], using SLCO, including verifying and simulating it.

To verify and simulate the SLCO model, and execute it on a LEGO® Mindstorms® robot, Engelen investigated transforming SLCO into other languages that could be used for these purposes. To prepare SLCO models for transforming them into these other languages, he first performed several transformations on the models from SLCO to itself. We implemented these transformations in our DSTL and we compare them with implementations in GPTLs to answer our research question.

3 Using a DSTL in practice

To investigate when DSTLs are useful in practice, we developed a DSTL for SLCO as a case study. We call this DSTL **SLCOtrans**. To validate the expressiveness of SLCOtrans, we implemented the transformations from **Section 3.5.1: 'Endogenous Transformations'** in the PhD Thesis of Luc Engelen [13]. By implementing the transformation in SLCOtrans and comparing them with implementations in GPTLs[2], we show in which cases DSTLs are convenient, and in which cases their limitations are reached.

3.1 Exogenous transformations

An *exogenous* transformation is a transformation between two different languages. In his thesis, Luc Engelen presents several exogenous transformations from SLCO to other languages, for simulation, execution, and verification.

For *simulation* of SLCO models, SLCO can be transformed to **POOSL**: "a formal modeling language for simulation and performance analysis" [13, 23].

For *execution* on a LEGO® Mindstorms® [16] robot, SLCO can be transformed to **NQC** [24]. "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" [13].

For *verification*, SLCO can be transformed to **Promela**. Promela is a language used for a model checker called SPIN [6, 7]. "[SPIN] can, among others, check a model for deadlocks, unreachable code, and determine whether it satisfies a Linear Temporal Logic (LTL) property [25]" [13].

Often, the languages to which we want to transform SLCO *do not support all the concepts available in SLCO*. For example NQC only supports asynchronous communication and POOSL only supports synchronous communication, whereas SLCO supports both.

Also, there sometimes are *practical limitations* on the target platform that do not exist in SLCO. For example, the number of concurrent objects in NQC are limited by the number of Mindstorms® microprocessors available, because each concurrent object has to run on a separate microprocessor.

For a more extensive overview of the limitations of the languages we want to transform SLCO to, see **Section 3.4: 'Semantic Gaps and Platform Gaps'** of the PhD Thesis of Luc Engelen [13].

3.2 Endogenous transformations

Before transforming an SLCO model into a different language, it is useful to transform the SLCO model to a different SLCO model that abides by the limitations of the eventual target language or platform. It increases modularity and reusability. A transformation where the source and target languages are the same is called an *endogenous* transformation.

Several endogenous transformations are helpful to make SLCO models suitable for the eventual target languages. We now present an overview of the endogenous transformations presented in **Section 3.5** of the PhD Thesis of Luc Engelen [13], with short annotations.

We have implemented these transformations in our DSTL, SLCOtrans, and compared the implementations with implementations in GPTLs[2]. This is discussed in **Section 8: 'Extending the design'**. Short explanations of the transformations are also given there. Longer explanations and

[2] Though the implementations are available in Xtend [38], we present them in this paper using pseudocode, so they are more readable and the reader does not require knowledge of Xtend.

more information can be found in the PhD Thesis of Luc Engelen [13]. Concepts in SLCO that are used in the following overview (such as *objects* and *channels*) are explained in **Section 4: 'SLCO'**.

We implemented the following endogenous transformations:

- Synchronized Communication over Asynchronous Channels.
  - Simple. With acknowledgment signals for synchronization. Only for restricted models.
  - General. For states with multiple outgoing transitions.
- Lossless Communication over a Lossy Channel.
  (with Concurrent Alternating Bit Protocol (CABP).)
- Adding Delays to Transitions.
- Replacing Strings by Integers.
- Making the Sender of a Signal Explicit.
  (By adding channel index to signal names. For broadcast in LEGO® Mindstorms®.)
- Reducing the Number of Objects: merging.
  (Replace unidirectional synchronous channels with shared vars.)
- Making all Signal Names Equal: rename signals.
  (For use with CABP. Include original name as argument.)
- Replacing a Bidirectional Channel by two Unidirectional Channels.
- Exclusive Channels for Pairs of State Machines.
  (2 x 2 state machines gives 4 channels: each state machine to each state machine.)
- Reducing the Number of Channels: merging.
- Cloning Classes. (auxiliary.)
- Removing Unused Classes. (auxiliary.)

This list is presented here to give an early, short overview of the transformations we implemented. It gives an idea of the kinds of transformations we implemented. It is not necessary yet to understand exactly what they do yet. This will be explained in **Section 8: 'Extending the design'**.

3.3 Comparing our DSTL to GPTLs

We use the following *methodology* to investigate whether DSTLs are useful in practice.

First, in **Section 7: 'The limitations of our DSTL'**, we explain some cases in which the use of SLCOtrans (and DSTLs in general) is limited, and we discuss a general problem with DSTLs.

Then, in **Section 8: 'Extending the design'**, we compare implementations of the transformations in SLCOtrans with implementations in (a notation that is representative of) traditional model transformation languages. We compare the conciseness of the implementations, how difficult it is to implement the transformations, and how easy to understand the implementations are. We explain for each transformation which aspect of feature of SLCOtrans makes them more or less concise. We then discuss for which transformations using SLCOtrans provides benefits, and explain the specific advantages that SLCOtrans provides in these cases.

4 SLCO

We created our DSTL for SLCO, because it is a simple but nontrivial language, that is advanced enough to be used for practical applications. In **Section 3.1: 'Exogenous transformations'** we have mentioned that SLCO can be simulated, executed, and verified, by transforming it to other languages. In this section, we will give a short introduction into SLCO. A more extensive explanation can be found in the documentation of SLCO [12, 13].

SLCO is short for Simple Language of Communicating Objects. SLCO can specify systems consisting of objects that operate in parallel and communicate. The behavior of these objects is specified with state machines with extensions specific to SLCO. SLCO has a graphical (e.g. **Figure 2**) and a textual notation (e.g. **Code fragment 1**, e.g. **Code fragment 2**). Part of the abstract syntax of SLCO can be seen in **Figure 3**.

An SLCO model consists of *objects*, *classes*, and *channels*. Objects are instances of classes. Objects contain *ports*, *variables*, and *state machines*. Objects communicate through their ports over channels. Channels can be *unidirectional* or *bidirectional*. Channels can support *synchronous* or *asynchronous* communication. Asynchronous channels can be *lossless* or *lossy*. State machines within an object can communicate with each other through shared variables. The transitions of state machines can contain *statements*. Statements can *assign a value* to a variable, *send a signal* to a port, wait until a certain *signal is received* from a port, *wait for a condition* to occur, and *wait for a specified amount of time*. Signals can have a number of *arguments*, with values of type boolean, integer, or string.

Figure 2 shows the graphical notation of SLCO. The top left shows an object `p` of class `P` and an object `q` of class `Q`. Object `p` has ports `In1`, `In2`, and `InOut`. Object `q` has ports `Out1`, `Out2`, and `InOut`. Between them are channels `c1`, `c2`, and `c3`. `c1` has an argument of type Boolean, `c2` has an argument of type Integer, and `c3` has an argument of type String. `c1` is a unidirectional asynchronous lossless channel, `c2` is a unidirectional asynchronous lossy channel, and `c3` is a bidirectional synchronous channel.

The top right of **Figure 2** shows that class `P` contains a variable `m` of type Integer with initial value 0 and state machines `Rec1`, `Rec2`, and `SendRec3` which contains a variable `s` of type String, and that class `Q` contains state machine `Com`, which also contains a variable `s` of type String.

The bottom of **Figure 2** shows the states and transitions in the state machines, and statements in the transitions. The statement `receive P([[false]]) from In1` means: 'receive signal `P` from port `In1` if its (only) argument equals `false`'.

The statement `receive Q(m | m >= 0) from In2; m := m + 1` means: 'receive signal `Q` from port `In2` if its argument is higher than or equal to 0, and then increase variable `m` by 1'.

The statement `m == 6` continues only when variable `m` equals 6. The statement `send S("a") to InOut` means: 'send signal `S` with string value `"a"` as argument to port `InOut`'. The statement `after 5 ms` means the transition waits 5 milliseconds. The rest of the statements is similar to those already mentioned.

Code fragment 2 shows the same model as **Figure 2**, but in the textual notation, with some parts left out. **Code fragment 1** shows a different model, in which two objects send signals back and forth.

```
model PingPongModel {
  classes
    Ping {
      ports
        P

      state machines
        Ping {
          initial
            SendState

          state
            ReceiveState

          transitions
            SentToReceive from SendState to ReceiveState {
              send Ping() to P
            }

            ReceiveToSend from ReceiveState to SendState {
              receive Pong() from P
            }
        }
    }

    Pong {
      ports
        P

      state machines
        Pong {
          initial
            ReceiveState

          state
            SendState

          transitions
            ReceiveToSend from ReceiveState to SendState {
              receive Ping() from P
            }

            SentToReceive from SendState to ReceiveState {
              send Pong() to P
            }
        }
    }

  objects
    Pi : Ping
    Po : Pong

  channels
    Producer_To_Consumer() sync between Pi.P and Po.P
}
```

Code fragment 1: *A model in the textual form of the DSL called SLCO. Object* `Pi` *of class* `Ping` *sends signal* `Ping()` *to object* `Po` *of class* `Pong` *over channel* `Producer_To_Consumer`*. When the signal is received,* `Po` *can send signal* `Pong()` *back to* `Pi`*. After* `Pi` *receives* `Pong()`*, it can send* `Ping()` *again to* `Po`*, and the cycle continues.*

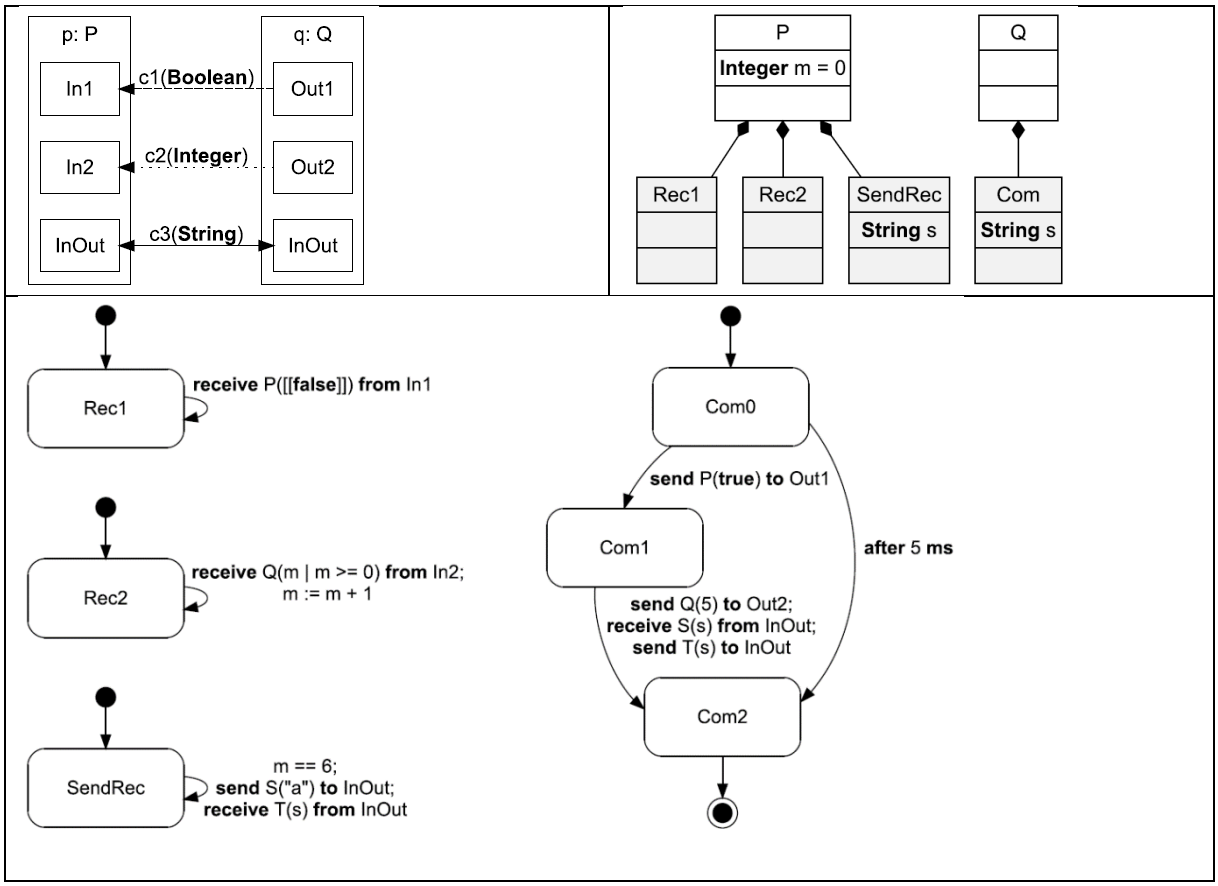

Figure 2: Elements from the graphical notation of SLCO. *This notation is explained in* ***Section 4: 'SLCO'****.* ***Upper left:*** *objects, ports, and channels.* ***Upper right:*** *classes, state machines, and variables.* ***Bottom:*** *state machines.*

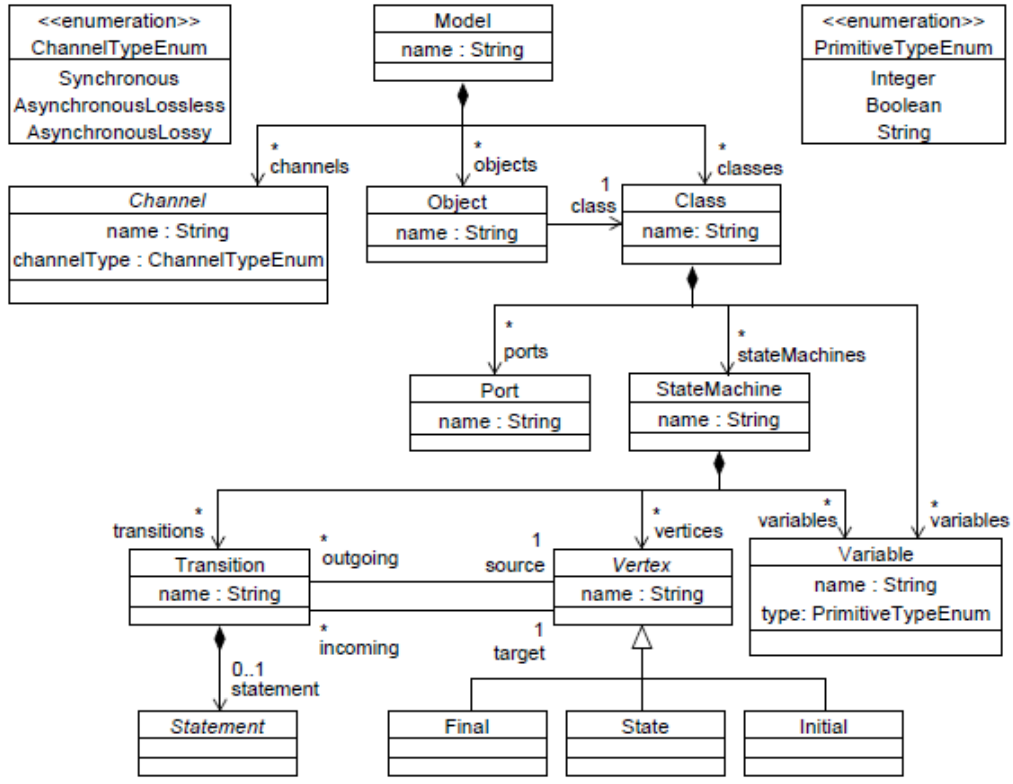

Figure 3: *Part of the* ***abstract syntax/metamodel of SLCO*** *containing the main constructs of the language. See the documentation of SLCO [12, 13] for the rest of the metamodel.*

```
model CoreWithTime {
  classes
    Q {
      variables
        Integer m = 0

      ports
        Out1 Out2 InOut

      state machines
        Com {
          variables
            String s

          initial Com0

          state Com1 Com3 Com4

          final Com2

          transitions
            InitialToState from Com0 to Com1 {
              send P(true) to Out1
            }
            ...
        }
    }
    ...
  objects
    p : P
    q : Q

  channels
    c1(Boolean) async lossless from q.Out1 to p.In1
    c2(Integer) async lossy from q.Out2 to p.In2
    c3(String) sync between p.InOut and q.InOut
}
```

Code fragment 2: *Part of a textual SLCO model.*

5 Initial design

We now present the initial design of SLCOtrans. In **Section 6: 'Implementation'**, we explain how we implemented it. In **Section 8: 'Extending the design'**, we improve and extend the design.

The Xtext [26, 27] grammar of SLCOtrans is shown in **Code fragment 7** (pp. 26-27). We will explain SLCOtrans however based on an example transformation, shown in **Code fragment 3**. The transformation is one of those presented in the PhD thesis of Luc Engelen [13]. The transformation converts *bidirectional* channels to two *unidirectional* channels: one in each direction. Furthermore, it splits the ports that the bidirectional channels were connected to into two ports, one to send signals over the outgoing unidirectional channel and one to receive signals from the incoming unidirectional channel. It also makes sure signals are sent to and received from the new ports. We will call this transformation '*Bi2Uni*'.

Code fragment 4 shows a model in SLCO. **Code fragment 5** shows what the SLCO model in **Code fragment 4** looks like after the transformation.

The initial design of SLCOtrans does not support all transformations presented in the PhD thesis of Luc Engelen [13]. It is designed to at least support (all parts required for) the *Bi2Uni* transformation. Therefore, it can transform ports, channels, and transitions and the statements associated with them.

The transformation in **Code fragment 3** starts with a list of *ports*, containing the old ports and the new ports. The old ports are automatically removed if they are not used anymore after the transformation. This is not always the case, for example when in the input model, there is also a unidirectional channel connected to the same port as the bidirectional channel being transformed. In that case, the unidirectional channel remains connected to the port after the transformation.

Then, the transformations on the *channels* are specified. The channels in the 'match'-block are replaced by the channels in the 'add'-block. The block is called 'add' because the old channels are only removed when they are not used anymore at the end of the transformation. This means they can still be referred to during the transformation.

The 'match'- and 'add'-blocks work by pattern matching. The 'match'-block matches a synchronous, bidirectional channel. It stores the names of the ports it is connected to in *transformation variables*[3] `Pi` and `Po` during the transformation. Each matched channel is replaced by two unidirectional channels. Each occurrence of a transformation variable is replaced by the name in the source SLCO model that it was matched with. If the transformation variable is part of a longer string, the name used in the output SLCO model is the name used in the transformation file, with the transformation variable in it replaced by the name in the source SLCO model.

This is just a convenience though. For the name of the channel no advanced name matching between the match and replacement is used, but instead it is just suffixed by a number. Most important is that *elements (such as ports) with the same transformation variable in the SLCOtrans transformation match the same element (such as a port) in the source- (and target) SLCO file*, regardless of their name there.

[3] To give a concise definition: a *transformation variable* is a name in an SLCOtrans file that can match something in the SLCO model on which the transformation is performed. Multiple occurrences of the transformation variable refer to the same matched element in the SLCO model.

In channels, it is important though which replacement port results from which matched port, because the object to which the resulting channel is connected is the same one as the one in which the original matched port was contained.

We will now show how the rules so far apply to our example. When the transformation in **Code fragment 3** is applied to **Code fragment 4**, resulting in **Code fragment 5**, then `PingPong() sync between Pi and Po` matches `Producer_To_Consumer() sync between Pi.P and Po.P`. It is replaced by `PingToPong() sync from Pi_send to Po_receive` and `PongToPing() sync from Po_send to Pi_receive` in the SLCOtrans file, which means it is replaced by `Producer_To_Consumer1() sync from Pi.P_send to Po.P_receive` and `Producer_To_Consumer2() sync from Po.P_send to Pi.P_receive` in the SLCO file.

For example, `Pi_send` in the SLCOtrans file is derived from `Pi` in `PingPong() sync between Pi and Po` in the SLCOtrans file. Because that `Pi` matches port `P` in object `Pi` [4] in the input SLCO model, `Pi_send` in the SLCOtrans file becomes port `P_send` in object `Pi` in the output SLCO model, denoted as `Pi.P_send`. So `Pi` in `Pi_send` is replaced by `P`, which results in `P_send`. And also the object that it belongs to is made the same.

`Pi_receive` too is derived from `Pi` in `PingPong() sync between Pi and Po`. Therefore `Pi_receive` in the SLCOtrans file becomes `P_receive` in the output SLCO model.

Now we will continue explaining the rest of the SLCOtrans example. After the ports and channels, the transformations on *state machines* are specified. There are two in this example: one to replace the port matched by transformation variable `Pi` by the new ports in **`send`** and **`receive`** statements (`Pi.P_send` in **`send`** statements and `Pi.P_receive` in **`receive`** statements), and one to do the same for the port matched by transformation variable `Po`.(for which the new ports are `Po.P_send` and `Po.P_receive`).

We will now look closer at the first state machine transformation. The transformation matches two transitions: one from the state matched by transformation variable `SendState` to the state matched by transformation variable `ReceiveState`, and one between the same states but in the reverse direction. The first one contains a **`send`** statement to the port matched by transformation variable `Pi`. The second one contains a **`receive`** statement from the same port. The names of the sent and received signals are matched by transformation variables `Ping` and `Pong`. This is relevant again in the replacement.

The two matched transitions are replaced by two transitions between the same two states as those in the original match (because the same transformation variables `SendState` and `ReceiveState` are used). The **`send`** or **`receive`** statement in them also stays the same as well as the name of the signal involved (because the same transformation variables `Ping` and `Pong` are used). The port over which the signals are communicated however are replaced by the new ports that replace the port matched by transformation variable `Pi`: the ports indicated by transformation variables `Pi_send` and `Pi_receive`, which refer to `Pi.P_send` (port `P_send` in object `Pi`) and `Pi.P_receive` in the (output) SLCO model.

The other state machine transformation is similar.

[4] NB: `Pi` *in the SLCO model is* ***not*** *the same as* `Pi` *in the SLCOtrans file*: `Pi` in the SLCOtrans file is a transformation variable, which matches a port (and its object) in this case, while `Pi` in the SLCO model is the name of an object.

5.1 Graph rewriting

The technique behind the transformations in SLCOtrans that we just mentioned, is called *graph rewriting* [28]. The terminology comes from category theory. Two common approaches to graph rewriting are *double-pushout* and *single-pushout* [29].

Both replace all occurrences of a pattern graph in a host graph with a replacement graph. In other words, the pattern graph is 'cut out' and the replacement graph is 'glued back in'. For double-pushout though, an extra graph, called interface graph or gluing graph, is used as an interface. The gluing graph indicates nodes and edges that have to be preserved during a replacement. For double-pushout an extra condition has to hold for executing a replacement: the gluing condition [30]. The gluing condition consists of two parts: the dangling condition and the identification condition.

The *dangling condition* [31] states that an occurrence of the pattern graph in the host graph can only be replaced if no edges are left 'dangling' without source or target node after 'cutting out' the pattern graph.

The *identification condition* [31], for which the gluing graph is used, states that a match is only valid if the matched nodes also appear in the gluing graph (and are thus preserved).

Single-pushout is more powerful than double-pushout, but also more dangerous, as it can leave an invalid graph after the transformation (with dangling edges). We opted for double-pushout in our design.

The `glue` block indicates which states are in the gluing graph.

```
model transformation {

    ports
        // Old ports (if the ports are no longer in use after the transformation the transformation tool
        // should remove the unused ports)
        Pi
        Po
        // New ports (the transformation tool should introduce these in the right places)
        Pi_send
        Pi_receive
        Po_send
        Po_receive

    channels
        // Channels that are no longer used after transformation should be removed by the transformation
        // tool.
        match {
            PingPong() sync between Pi and Po
        }
        add {
            PingToPong() sync from Pi_send to Po_receive
            PongToPing() sync from Po_send to Pi_receive
        }

    transformations
        state machine transformation {
            glue
                SendState
                ReceiveState

            match {
                transitions
                    from SendState to ReceiveState {
                        send Ping() to Pi
                    }

                    from ReceiveState to SendState {
                        receive Pong() from Pi
                    }
            }
            replace with {
                transitions
                    from SendState to ReceiveState {
                        send Ping() to Pi_send
                    }

                    from ReceiveState to SendState {
                        receive Pong() from Pi_receive
                    }
            }
        }

        state machine transformation {
            glue
                SendState
                ReceiveState

            match {
                transitions
                    from ReceiveState to SendState {
                        receive Ping() from Po
                    }

                    from SendState to ReceiveState {
                        send Pong() to Po
                    }
            }
            replace with {
                transitions
                    from ReceiveState to SendState {
                        receive Ping() from Po_receive
                    }

                    from SendState to ReceiveState {
                        send Pong() to Po_send
                    }
            }
        }
}
```

Code fragment 3: A transformation in our DSTL, SLCOtrans. *It converts* ***bidirectional*** *channels* ***to*** *two* ***unidirectional*** *channels: one in each direction. (It also splits the ports the channel is connected to).*

```
model PingPongModel {
  classes
    Ping {
      ports
        P

      state machines
        Ping {
          initial
            SendState

          state
            ReceiveState

          transitions
            SentToReceive from SendState to ReceiveState {
              send Ping() to P
            }

            ReceiveToSend from ReceiveState to SendState {
              receive Pong() from P
            }
        }
    }

    Pong {
      ports
        P

      state machines
        Pong {
          initial
            ReceiveState

          state
            SendState

          transitions
            ReceiveToSend from ReceiveState to SendState {
              receive Ping() from P
            }

            SentToReceive from SendState to ReceiveState {
              send Pong() to P
            }
        }
    }

  objects
    Pi : Ping
    Po : Pong

  channels
    Producer_To_Consumer() sync between Pi.P and Po.P
}
```

Code fragment 4: *A model in SLCO* before *the* Bi2Uni *transformation.*

```
model PingPongModel {
  classes
    Ping {
      ports
        P_send
        P_receive

      state machines
        Ping {
          initial
            SendState

          state
            ReceiveState

          transitions
            SentToReceive from SendState to ReceiveState {
              send Ping() to P_send
            }

            ReceiveToSend from ReceiveState to SendState {
              receive Pong() from P_receive
            }
        }
    }

    Pong {
      ports
        P_receive
        P_send

      state machines
        Pong {
          initial
            ReceiveState

          state
            SendState

          transitions
            ReceiveToSend from ReceiveState to SendState {
              receive Ping() from P_receive
            }

            SentToReceive from SendState to ReceiveState {
              send Pong() to P_send
            }
        }
    }

  objects
    Pi : Ping
    Po : Pong

  channels
    Producer_To_Consumer1() sync from Pi.P_send to Po.P_receive
    Producer_To_Consumer2() sync from Po.P_send to Pi.P_receive
}
```

Code fragment 5: *The SLCO model of* ***Code fragment 4*** *after the* Bi2Uni *transformation. Changes have been marked yellow.*

6 Implementation

SLCOtrans consists of two parts: the syntax and the semantics. We have defined the syntax in Xtext [26, 27]. The syntax is shown in **Code fragment 7**. We have explained it in **Section 5: 'Initial design'**.

The semantics have been implemented by writing an *interpreter* in EOL [18, 19] (from the Epsilon family [20, 21, 22]). The code of the interpreter is shown in **Code fragment 41** in **'Appendix A: The implementation of SLCOtrans'**. We will explain this in **Section 6.1: 'In-depth explanation'**.

The interpreter uses an SLCO model and an SLCOtrans file as input, and outputs an SLCO model on which the transformations in the SLCOtrans file have been performed. This process is shown in **Figure 6**.

Another option would be to create a program generator. The program generator would then input the SLCOtrans file and output a transformation in a GPTL (or other language). The generated transformation in the GPTL could then be applied to the SLCO model. The generated transformation would then input the SLCO model and output the transformed SLCO model.

We chose to create an interpreter, because for a program generator, additionally to the metamodel for SLCO and SLCOtrans, the metamodel of the language to generate to (the GPTL) would need to be constructed, in case of a model-to-model transformation [32].

We initially tried to implement our interpreter in the general-purpose transformation language ETL (the Epsilon Transformation Language [33, 34]) instead of EOL, which is a model-oriented language, but not specific to transformations. EOL is a subset of ETL. *We switched from ETL to EOL* because some main features of ETL that are specific to transformations, do not support transformations on multiple input objects at the same time. This was something we often needed (as we combine two input models -a SLCO model and a SLCOtrans model- into one output model: the transformed SLCO model).

We used Eclipse Modeling Tools [35], with Xtext [26, 27] and Epsilon [20, 22]. Eclipse Modeling Tools is based on the Eclipse Modeling Framework [36, 37] (a framework to support using models in Eclipse).

6.1 In-depth explanation

6.1.1 ANT

In addition to the interpreter in EOL, an ANT-script [38] is needed, to indicate the parameters with which the interpreter is executed and the location of the SLCOtrans and SLCO models used. An example of an ANT-script that can be used is shown in **Code fragment 8**. The values of the attributes `modelFile` have to be changed to the location of respectively the SLCO model to perform the transformation on and the SLCOtrans model. This script is used for an in-place transformation. An in-place transformation is a transformation where the input model is replaced by the output model.

The filename extension of the ANT-script should be '.ant' or '.xml'. The ANT-script can be executed as follows: right click on it. Then click 'Run as'->'(2) ANT build…'. In the tab 'JRE' select 'Run in the same JRE as the workspace'. Then click 'Run'. From then on it can also be executed by using 'Run as'->'(1) ANT build' in the context menu of the ANT-file.

The ANT-script lets the interpreter perform the transformation in the file indicated in the ANT-script on the SLCO model in the file indicated in the ANT-script.

6.1.2 EOL

The implementation of the interpreter in EOL consists of two main parts. The first part (lines 19-249, in **Code fragment 41** in **'Appendix A: The implementation of SLCOtrans'**) transforms the channels and the second part (lines 249-358) transforms the state machines.

In these parts the ports that are still used, and therefore need to be kept, are stored in the `OrderedSet` `portsKeep`. At the end of the transformation (lines 359-363) all ports that are not used anymore (and therefore are not in `portsKeep`) are removed.

At the beginning (lines 5-16), some abbreviations are introduced for the models used as input and output to the transformation, that is: the SLCO model(s) and the SLCOtrans model. Some other abbreviations for commonly used constructs are also introduced.

In the part about transforming channels, the following happens. It is checked whether a pattern of channels in the SLCO input model matches a pattern of channels in the 'match'-block of the SLCOtrans model. This happens when for a selection of channels from the SLCO model two things are true. It should hold that for every channel in that selection there is a corresponding channel in the 'match'-block of the SLCOtrans model with the same channel type and number and types of arguments. And it should hold as well for every channel in that selection, that if a port is the same one (so also in the same class) as a port in another one of those channels, that then the same holds for the corresponding channels in the 'match'-block of the SLCOtrans model.

Then, if a match is found, it is replaced by a structure[5] of channels corresponding to the structure of channels in the 'add'-block of the SLCOtrans model. This means that ports in the structure of channels in the SLCO output model are replaced by the ports in the SLCO input model that correspond to the transformation variables at the same place in the structure of channels in the 'add'-block of the SLCOtrans model. So the replacement in the SLCO output model is the SLCOtrans model with the transformation variables replaced by their values, (ports in this case). And these values are determined by which transformation variables in the 'match'-block of the SLCOtrans model match which ports in the SLCO input model.

An example of how elements in the SLCO output model follow from the elements in the SLCO input model and the transformation in the SLCOtrans file is shown in **Figure 4**.

SLCO input	`Channel1A() sync between objA.portA and objB.portB` `Channel1B() async lossless between objA.portC and objB.portD`
SLCOtrans match	`Channel1X() sync between portX and portY`
SLCOtrans replacement	`Channel1Y() sync from portY to portX`
SLCO output	`Channel1A1() sync from objA.portB to objB.portA` `Channel1B() async lossless between objA.portC and objB.portD`

Figure 4: *An example of matching and replacing with exact name matches in the SLCOtrans file.*

We will now provide more detail. First (line 40), the new name of each channel is assigned: the name of the corresponding input channel (if possible), followed by a number. Then (lines 42-56), the argument types and channel type are copied from the replacement in the SLCOtrans model to the replacement in the SLCO output model.

[5] By a structure of channels we mean a collection of channels that might be connected to the same port(s) as other channels in the collection. These other channels might be connected to the same port(s) as even other channels in the collection, etcetera.

After that (lines 70-236), for each transformation variable (for a port) in the 'replace'-block it is checked whether it matches a transformation variable (for a port) in the 'match'-block. If it is an exact match (e.g lines 76-81), the port entered into the SLCO output model is the one in the SLCO input model corresponding to the transformation variable.

If it is not an exact match, but the name of the transformation variable in the 'match'-block is part of the name of the transformation variable in the 'replace'-block (e.g. lines 82-107), then a new port is created (if it does not yet exist), where the name of the transformation variable in the 'match'-block in the name of the transformation variable in the 'replace'-block is replaced by the name of the corresponding port in the SLCO input model.

An example of how elements in the SLCO output model follow from the elements in the SLCO input model and the transformation in the SLCOtrans file in the case the replacement names in the SLCOtrans file are not exact matches to the match names is shown in **Figure 5**.

SLCO input	ChannelA() sync between objA.portA and objB.portB ChannelB() async lossless between objA.portC and objB.portD
SLCOtrans match	ChannelX() sync between portX and portY
SLCOtrans replacement	ChannelY() sync from pre_portY__Post to PREportX_post
SLCO output	ChannelA1() sync from objA.pre_portB__Post to objB.PREportA_post ChannelB() async lossless between objA.portC and objB.portD

Figure 5: *An example of matching and replacing with inexact name matches in the SLCOtrans file.*

Approximately the same thing as for channels happens for state machines (lines 249-358). But instead of channels, transitions and states are matched and replaced, and the statements associated with transitions.

We have not yet implemented all parts of fully generic transformations. So not everything that can be expressed in SLCOtrans can be executed by the interpreter. For example, channels can only be replaced by unidirectional channels for now. Otherwise, there are no further restrictions on the replacement channels. The amount of replacement channels is unrestricted. Also, the channels can have any communication type (synchronous, synchronous lossy, and synchronous lossless) and they can support any amount of arguments of any type specified in the Xtext grammar.

The implementation is still quite limited for transforming state machines. It can do little more than execute the transformations for state machines in **Code fragment 6** (p. 19). An example of something it can do however, is connecting the signal communication statements to ports with arbitrary names.

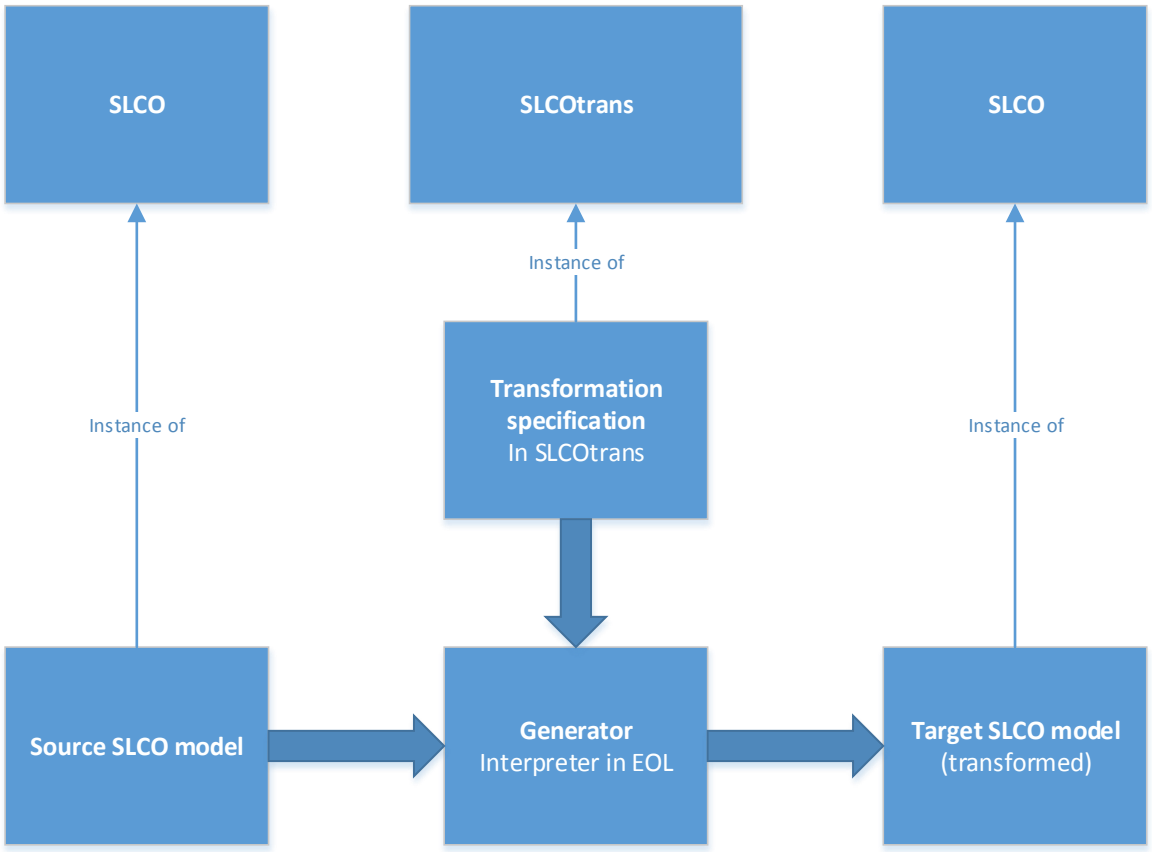

Figure 6: How our implementation of SLCOtrans is executed. *A transformation written in SLCOtrans is read by an interpreter written in EOL. The transformation instructions read from the SLCOtrans file are applied to the source model which results in the target model.*

```
grammar org.xtext.textualslcotrans.TextualSlcoTrans with org.eclipse.xtext.common.Terminals

generate textualSlcoTrans "http://www.xtext.org/textualslcotrans/TextualSlcoTrans"

ModelTransformation returns ModelTransformation:
    {ModelTransformation}
    'model transformation'
    '{'
        ('ports' (ports+=Port)*)?
        ('channels'
            ('match' '{' (channelsL+=Channel)* '}')?
            ('add' '{' (channelsR+=Channel)* '}' )?
        )?
        ('transformations' (transformations+=Transformation)* )?
    '}';

Transformation returns Transformation:
    StateMachineTransformation;

StateMachineTransformation :
    'state machine transformation' '{'
        'glue' glueStates+=Glue (glueStates+=Glue)*    // Glue states
        'match' stateMachineL=StateMachine             // Left SM pattern
        'replace with' stateMachineR=StateMachine      // Right SM pattern
    '}';

Channel :
    BidirectionalChannel | UnidirectionalChannel;

BidirectionalChannel :
    name = ID '(' (argumentTypes += ArgumentType (',' argumentTypes += ArgumentType)*)? ')'
    channelType = ChannelTypeEnum 'between'
    /*object1 = [Object] '.'*/ port1 = [Port] 'and' /*object2 = [Object] '.'*/ port2 = [Port];

UnidirectionalChannel :
    name = ID '(' (argumentTypes += ArgumentType (',' argumentTypes += ArgumentType)*)? ')'
    channelType = ChannelTypeEnum 'from' /*sourceObject = [Object] '.'*/ sourcePort = [Port] 'to'
/*targetObject = [Object] '.'*/ targetPort = [Port];

StateMachine :
    {StateMachine}
    '{'
        ('variables' (variables+=Variable)* )?
        ('glue' (vertices+=Glue)* )?
        ('state' (vertices+=State)* )?
        ('final' (vertices+=Final)* )?
        ('transitions' (transitions+=Transition)* )?
    '}';

Vertex :
    Final | Glue | State;

Final :
    name=ID;

Glue :
    name=ID;

State :
    name=ID;

Transition :
    'from' source=[Vertex] 'to' target=[Vertex] '{'
        (statements += Statement (';' statements += Statement)*)?
    '}';

Variable :
    type=PrimitiveTypeEnum name=ID ('=' initialValue=Expression)?;

Statement :
      Assignment
    | Expression
    | Delay
    | SendSignal
    | SignalReception;

Expression :
  TerminalExpression ({BinaryOperatorExpression.operand1 = current} operator = OperatorEnum operand2 =
Expression)?;

TerminalExpression returns Expression:
      BooleanConstantExpression
    | IntegerConstantExpression
    | VariableExpression
    | BracketExpression;

BracketExpression returns Expression:
  "(" Expression ")";
```

```
SignalArgument :
    SignalArgumentExpression | SignalArgumentVariable;

Assignment :
    variable=[Variable] ':=' expression=Expression;

Delay returns Delay:
    'after' value=INT 'ms';

BooleanConstantExpression :
    value = BOOLEAN;

IntegerConstantExpression :
    value=INT;

SendSignal :
    'send' signalName=ID '(' (arguments+=Expression (',' arguments+=Expression)* )? ')' 'to' port =
[Port];

SignalReception :
    'receive' signalName = ID '(' (arguments += SignalArgument (',' arguments += SignalArgument)*)? ('|'
condition = Expression)? ')' 'from' port = [Port];

VariableExpression :
    variable=[Variable];

SignalArgumentExpression :
    {SignalArgumentExpression}
    '[[' expression = Expression ']]';

SignalArgumentVariable :
    {SignalArgumentVariable}
     variable=[Variable];

ArgumentType :
    type=PrimitiveTypeEnum;

Object :
    name = ID ':' class = [Class];

Class :
    {Class}
    name = ID '{'
    ('variables' (variables += Variable)* )?
    ('ports' (ports += Port)* )?
    ('state machines' (stateMachines += StateMachine)* )?
    '}';

enum ChannelTypeEnum :
    async_lossless = 'async lossless' | async_lossy = 'async lossy' | sync = 'sync';

enum PrimitiveTypeEnum:
    Integer | Boolean; // | String;

Port :
    name=ID;

enum OperatorEnum :
    atLeast = '>=' | atMost = '<=' | add = '+' | and = '&&' | or = '||' | equals = '==' | differs = '!='
| subtract = '-';

terminal BOOLEAN :
  'true' | 'false';
```

Code fragment 7: *The concrete syntax of SLCOtrans v1, specified in the Xtext grammar language.*

```
<?xml version="1.0"?>
<project default="main">
  <target name="loadModels">
    <epsilon.emf.loadModel
    name     = "TextualSlco"
    modelFile    = "../../slco_ecore/pipotxt cp for transfo.slco2"
    metamodelUri = "http://www.xtext.org/TextualSlco"
    read     = "true"
    store        = "true"
    />
    <epsilon.emf.loadModel
    name     = "TextualSlcoTrans"
    modelFile    = "../../slcotrans_ecore/pipotrtxt cp for transfo.slcotrans2"
    metamodelUri = "http://www.xtext.org/textualslcotrans/TextualSlcoTrans"
    read     = "true"
    store        = "false"
    />
  </target>
  <target name="main" depends="loadModels">
    <epsilon.eol src="transform.eol">
        <model ref="TextualSlco"/>
        <model ref="TextualSlcoTrans"/>
    </epsilon.eol>
  </target>
</project>
```

Code fragment 8: *An ANT –script to pass the right arguments and location of the SLCOtrans and SLCO models used to the interpreter. The values of the attributes `modelFile` have to be changed to the location of respectively the SLCO model to perform the transformation on and the SLCOtrans model. NB: This script is used for an in-place transformation.*

7 The limitations of our DSTL

While designing and investigating SLCOtrans, we encountered several limitations. The principles behind these limitations are generalizable to other DSTLs.

Suppose that someone wants to transform a bidirectional channel into something (for example another channel), regardless of its channel type: synchronous, asynchronous lossy, or asynchronous lossless. Then it would be possible to introduce a wildcard at the place where the channel type would normally be indicated in the concrete syntax of SLCO.

Suppose however that someone wants to transform a channel into something (for example another channel), regardless of whether it is unidirectional or bidirectional. Then it already becomes more difficult what to do, because that is determined by the usage of '`between ... and ...`' for a bidirectional channel, and '`from ... to ...`' for a unidirectional channel. One could replace both keywords by wildcards, but this is not very intuitive. The statement to match for would then become for example '`PingPong() sync ? Pi ? Po`'.

Especially in combination with the earlier wish of disregarding channel type, the intention becomes quite unclear. In that case the statement to match for could become for example '`PingPong() ? ? Pi ? Po`'. Even worse, this does not look much like the concrete syntax of SLCO anymore at all. So that would defeat the point of using the concrete syntax.

Using the abstract syntax, transforming a channel into something, regardless of whether it is unidirectional or bidirectional, is much easier. One can simply match for a `Channel`, which is the supertype of `UnidirectionalChannel` and `BidirectionalChannel`. As a matter of fact, using the abstract syntax it is clear that one is matching for a channel, while using the concrete syntax this is not so clear at all. For example, in the abstract syntax notation used in **Section 8: 'Extending the design'**, one can match for a '`Channel(name: 'ch1', channelType: ChannelType.Synchronous)`'.

The same problem as for unidirectional and bidirectional channels also occurs for example for statements in transitions. If one wants to match any statement regardless of which kind it is, then using the abstract syntax one can match for a `Statement`, the supertype of the five kinds of statements in SLCO, but using the concrete syntax a (new) construction would have to be used to enumerate each of these kinds of statements (in the concrete syntax) as a possible match.

This problem is also generalizable to other DSTLs. For example, a metamodel of SQL [39] might contain (among others) Statements (e.g. `SELECT`, and `UPDATE`), Clauses (e.g. `WHERE`, and `HAVING`), and Operators (e.g. `AND`, `OR`, `LIKE`, and `IN`). If a user would now for example like to modify for each Statement the table it was applied on, the user could match for a Statement using the abstract syntax, whereas using the concrete syntax (i.e. in a DSTL) the user would have to match for each subclass of Statement separately.

Having to enter wildcards for everything one does not want to restrict is also not very convenient in itself. Compare for example **Code fragment 9** with **Code fragment 10**.

```
Transition(
  statements: [Delay(value: 10)]
)
```

Code fragment 9: *A transition with one delay statement of 10 ms, in an abstract syntax notation.*

```
from ? to ? {
  after 10 ms
}
```

Code fragment 10: *A transition with one delay statement of 10 ms, SLCOtrans (concrete syntax).*

```
Transition(
  statements: ['''after 10 ms''']
)
```
Code fragment 11: *A transition with one delay statement of 10 ms, in a notation combining abstract and concrete syntax.*

The notation in **Code fragment 9** means: 'the value of attribute `statements` of the (matched) `Transition` should be a list with in it only a `Delay` of which attribute "`value`" equals 10'. The square brackets denote a list comprehension.

In **Code fragment 10**, having to enter wildcards for everything one does not want to restrict is not so bad yet. But for elements where much more wildcards have to be entered, and if these elements occur many times in a transformation, this becomes quite cluttering and possibly confusing.

In **Code fragment 11**, a way to combine the abstract and concrete syntax is shown. Triple quotes surround the parts in concrete syntax. **Code fragment 11** demonstrates that for example in such cases, using concrete syntax in transformations *is* convenient.

While trying to apply extended (and improved) versions of SLCOtrans to all endogenous transformation in the PhD Thesis of Luc Engelen [13], we discovered that the main problem with using concrete syntax in transformations (without abstract syntax) is that many of the transformations (especially the more complex ones) are so granular that the concrete syntax becomes cluttered with enough things related to the specifics of the transformation under development, to significantly reduce the benefits that the concrete syntax provides. For example the transformation in **Section 8.12: 'Reducing the Number of Objects'** is so complex that it would be very difficult to understand for a domain expert, regardless of whether it were in concrete syntax or not. We discuss the transformations in the thesis of Engelen and our implementations of them in **Section 8: 'Extending the design'**.

There are few transformations in **Section 8** where the concrete syntax can be used well. A case when using the concrete syntax does show its strength however, is when large fragments of the DSL can be used with few changes. This happens for example in the transformation of a lossless to a lossy channel, while keeping the functional behavior the same. This is shown in **Section 8.9: 'Lossless Communication over a Lossy Channel'**.

8 Extending the design

In **Section 7** we have demonstrated the limitations of using concrete syntax in a transformation language. Our design of SLCOtrans also has another notable feature: it is based on pattern matching. In this section, we present an improved and extended version of SLCOtrans, to investigate the applicability of a transformation language based on pattern matching. We refer to this new version of SLCOtrans as **SLCOtrans v2**.

The process by which we extend SLCOtrans is by implementing transformations from the PhD Thesis of Luc Engelen [13] and add new features when we need them or think they can improve the implementation. Sometimes we use slightly different transformations. We compare the implementations with implementations in traditional GPTLs. Implementations of the transformations in Xtend [40] have been created by Luc Engelen. These are available at *http://www.win.tue.nl/~lengelen/SLCO.zip*. We present shorter implementations here in pseudocode though, so they can be understood without prior knowledge of Xtend and so the specifics of transformation languages are left out. This simplifies the code. Compare for example **Code fragment 13** in EOL, which has no 'prepend' operation, with **Code fragment 12** in pseudocode.

We give short explanations of each transformation we implement. Longer explanations can be found in **Section 3.5.1: 'Endogenous Transformations'** of the PhD thesis of Luc Engelen [13].

Before extending SLCOtrans while implementing transformations, we will first introduce some changes to the design of SLCOtrans in general.

We simplified the syntax. The user does not have to encapsulate his transformation in `model transformation {...}` anymore. We realized the separation between transforming channels and state machines was not necessary, so there now are only 'match'- and 'replace'-blocks in which everything can be transformed. These blocks are encapsulated by indentation instead of braces. The earlier 'add'-block is now merged into the new unified 'replace'-block. We believe that from the user perspective 'replace' is clearer than 'add', despite the origin in graph rewriting theory.

We also switched from double pushout graph rewriting to single pushout graph rewriting, because we thought double pushout unnecessarily complicated the syntax. So the 'glue'-block has been removed. We have explained single pushout and double pushout graph rewriting and their advantages and disadvantages in **Section 5.1: 'Graph rewriting'**.

8.1 Adding Delays to Transitions

The **Add Delays** transformation prepends each transformation with a delay of 10 ms.

Code fragment 14 shows the implementation in SLCOtrans v2. We will now explain the (new) concepts used in it.

- Everything in the 'match'-block(s) is replaced by everything in the 'replace with'-block(s). In other words, everything in the replace block is added and everything in the match block is removed.
- The match block searches for a `Transition` in the abstract model (i.e. abstract syntax or metamodel).
- For every match, the matched `Transition` is 'stored' in transformation variable `transit`.
- The value of attribute `statements` (the first one) of object `Transition` in `transit` is stored in `statements` (the second one). We will call the second '`statements`' a *label*. A label is also a transformation variable. At every place a transformation variable occurs, the value must be the same.

- In the replace block, every matched `Transition` is replaced by another `Transition`.
- The fragment '`copy transit into ..`' means that every attribute in transit is copied to an attribute that is named the same in the object it is copied into, if that attribute exists.
- Without '`copy transit into ..`' before it, '`Transition(..)`' means 'create a new transition' in the 'replace'-block.
- The attributes in '`Transition(..)`', -in this case just '`statements`'- indicate which (values of) attributes are overridden, in case there is a '`copy transit into ..`' before it.
- '`copy transit into ..`' is additionally used (but not here) to trace which match is replaced by which replacement.
- The `statements` property in the new `Transition` is replaced by a new `Delay` object with attribute `value` with value 10 list-concatenated with the content of the `statements` transformation variable.

Code fragment 14 (and **Code fragment 15**) match for a `Transition` and replace it with another `Transition` with the list of `Statement`s in the '`statements`' attribute prepended with a 10 ms `Delay` object.

For this transformation, SLCOtrans v2 (**Code fragment 14**) is not much more or less concise than a traditional language (**Code fragment 12**).

```
for transit in Transition.all:

    prepend Delay(value: 10) to transit.statements
```

Code fragment 12: *The* ***Add Delays*** *transformation in* ***pseudocode****.*

```
var Slco = TextualSlco;

for (transit in Slco!Transition.all) {

    var delay = new Slco!Delay;

    delay.value = 10;

    transit.statements = delay.asSequence.includingAll(transit.statements);
    // done this weird way because there is no prepend.
}
```

Code fragment 13: *The* ***Add Delays*** *transformation in* ***EOL****.*

```
match
    transit  = Transition(
        statements:  statements
    )

replace with
    copy transit into Transition(
        statements:  Delay(value: 10) + statements
    )
```

Code fragment 14: *The* ***Add Delays*** *transformation in* ***SLCOtrans v2*** *with a label for 'statements'.*

```
match
    transit  = Transition

replace with
    copy transit into Transition(
        statements:  Delay(value: 10) + transit.statements
    )
```

Code fragment 15: *The* ***Add Delays*** *transformation in* ***SLCOtrans v2*** *without a label for 'statements'.*

Code fragment 15 shows a slightly different way to implement the same transformation. In that example, attribute is not stored in transformation variable `transit` by using a label. Instead, in '`transit.statements`' attribute `statements` in transformation variable `transit` is directly retrieved.

Code fragment 16 shows another (naïve) way to implement the Add Delays transformation. This way is incorrect for this transformation however because it adds delays also at every level inside

expressions, because `Expression` is a subclass of `Statement` in the metamodel of SLCO. Although (therefore) this shorter approach does not work correctly for this transformation, it can be used in many other cases.

This implementation matches every place in the abstract model where 0 or more `Statement`s fit and if a place is found it stores every `Statement` found there in the list '`statements`'. The `Statement`s found at each place are replaced by the same `Statement`s , prepended by `Delay(value: 10)`. Matching 0 or more objects in the abstract model is indicated by an asterisk. Note that transformation variable `statements` can contain multiple matches: it automatically becomes a list because multiple results are stored in it.

```
// ERROR: also adds delay before expressions inside expressions.
match
    statements   = Statement*     // greedy. match any place where Statements* can fit.

replace with
    Delay(value: 10) + statements
```

Code fragment 16: *An incorrect version of the* ***Add Delays*** *transformation in* ***SLCOtrans v2****.*

8.2 Replacing Strings by Integers

The **Strings to Integers** transformation "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" [13].

Code fragment 17 shows the implementation in SLCOtrans v2. This implementation shows the use of two 'match'- and 'replace'-blocks. The part outside the 'match'- and 'replace'-blocks is procedural.

'`[]`' indicates an empty list. '`function index(str)`' is the function named '`index`' with parameter '`str`'. '`strs += str`' means 'add `str` to (the list) `strs`'. `PrimitiveType.String` and `PrimitiveType.Integer` are enum values. The language can not only match for *objects* in the abstract model, but also for values in enums. So, …

"	match
	  PrimitiveType.String
	replace with
	  PrimitiveType.Integer
		"

… means 'replace every value "`PrimitiveType.String`" that is found by the "match"-block by a "`PrimitiveType.Integer`"'. There can also be matched for other values like normal strings and integers. '`xx.lastIndex`' and '`index of … in …`' are built-in operations in SCLOtrans v2. '`xx.lastIndex`' returns the last index of a list and '`index of … in …`' returns the index at which an element occurs in a list.

'`index(str)`' in the second 'replace'-block means 'apply the function named "`index`" to the transformation variable "`str`"'. This means that the value returned by that function is used in that place instead of its argument.

```
strs = []            // list

function index(str):     // turn str into an int starting from 0.
    if str not in strs:
        strs += str              // add str to strs
        return strs.lastIndex         // IE: strs.length - 1
    else:
        return index of str in strs       // (findIndex)


match
    PrimitiveType.String      // this is an enum value.

replace with
    PrimitiveType.Integer


match
    StringConstantExpression(value: str)

replace with
    IntegerConstantExpression(value: index(str))
```

Code fragment 17: *The* ***Strings to Integers*** *transformation in* ***SLCOtrans v2****.*

In **Code fragment 17**, each occurrence of a string type indication is replaced by an indication for type integer (for example in the argument types of a channel). Also, each occurrence of a new string literal (a `StringConstantExpression` in the abstract model) is replaced by a unique integer (i.e. a `IntegerConstantExpression`), which we start numbering from 0. This converting to a number happens in the function '`index`'. Two occurrences of the same string literal should result in the same number. This is enforced as follows. Initially, an empty list of strings is created. Then, when the function '`index`' is called, the argument in parameter `str` is stored in the list, if it was not in there yet. Then, regardless of whether argument in `str` was already in the list, the number corresponding to the argument is returned. This is done by returning the index in the list at which the argument occurs.

```
list strs = []

function index(str):      // turn str into an int starting from 0.
    if str not in strs:
        strs += str               // add str to strs
        return strs.lastIndex         // IE: strs.length - 1
    else:
        return index of str in strs       // (findIndex)

////// turn str into an IntegerConstantExpression starting from 0:
function index(StringConstantExpression str):
    return IntegerConstantExpression(value: index(str.value))



for var in Variable.all:

    if var.type = PrimitiveType.String:

        var.type         = PrimitiveType.Integer
        var.initialValue = IntegerConstantExpression(value: index(var.initialValue))


for argType in ArgumentType.all:

    if argType.type = PrimitiveType.String:

        argType.type = PrimitiveType.Integer


for rcv in SignalReception.all:

    if rcv.condition = StringConstantExpression

        rcv.condition = IntegerConstantExpression(value: index(rcv.condition))


for thing in Assignment.all + SignalArgumentExpression.all + BinaryOperatorExpression.all:

    if thing.expression = StringConstantExpression

        thing.expression = IntegerConstantExpression(value: index(thing.expression))


for arg in SendSignal.all.arguments:  // shorthand

    if arg = StringConstantExpression

        arg = IntegerConstantExpression(value: index(arg))
```

Code fragment 18: *The* ***Strings to Integers*** *transformation in* ***pseudocode****.*

In the traditional (EOL-like) notation, in **Code fragment 18**, it is not possible to simply replace an object at all places it occurs. In order to do this, all references to the object need to be separately replaced. For this, all objects with a reference to the object are needed. So in the last 3 loops of **Code fragment 18**, all objects with a reference to `StringConstantExpression` are separately collected and the reference is replaced the new `IntegerConstantExpression` (in the attributes `.condition`, `.expression`, and `.arguments`). `SendSignal.all.arguments` is shorthand for looping through each element in attribute `.arguments` in each `SendSignal`.

Also, it is not possible to return all occurrences of an enum value, as opposed to *returning all occurrences of an object*: the construction in traditional languages used to simulate matching in SLCOtrans v2. So also in these cases the objects containing the references need to be separately collected and the references replaced. This happens in the first two loops.

These two differences cause SLCOtrans v2 to be significantly more concise than traditional languages for this transformation.

8.3 Making the Sender of a Signal Explicit

The **Identify Channels** transformation identifies the channel that a signal is sent over by suffixing the signal name with an underscore followed by a number uniquely identifying that channel. When channels are merged and signals have the name and number of arguments , then the sender can be derived by identifying the original channel.

Code fragment 19 shows the implementation of the transformation in SLCOtrans v2. '`(SendSignal or SignalReception)*`' means '(match) 0 or more objects that are of type `SendSignal` or of type `SignalReception`'. `ch` matches a `Channel`. This can be a `UnidirectionalChannel` or a `BidirectionalChannel`., because `Channel` is their superclass in the abstract model. '`port: ch.port1 or ch.port2 or ch.sourcePort or ch.targetPort`' means that matches are returned where attribute `port` of a `SendSignal` or `SignalReception` is one of those values. `UnidirectionalChannel`s has attributes `sourcePort` and `targetPort`, while `BidirectionalChannel` has attributes `port1` and `port2`. If an attribute is invalid, such as can occur in the construction of `or` expressions used here, then that part of the construction is ignored.

Transformation variable `sigs` is a list here (because the value stored in it is (a list of) multiple matches). So '`copy sigs into …`' copies a list into something. '`copy xx into …`', where `xx` is a list results in an iteration of '`copy … into …`' over each element in the list. Occurrences of `xx` in the rest of the statement now result in the current element of the list in the iteration. `yy._type` is a built-in operation of SLCOtrans, which returns the type of `yy`. So, in this case of `sigs._type` it refers to the type of the current element in the iteration. The asterisk indicates that `sigs` is copied into multiple 'targets'.

'`signalName: sigs.signalName + '_' + index(ch.name)`' means here that the attribute `signalName` of the newly created objects becomes appended with an underscore and the name of the channels converted into a unique integer. The function '`index`' is the same one as in **Section 8.2: 'Replacing Strings by Integers'**. '`ch`' in the 'replace'-block indicates that the channel stored in `ch` is preserved.

```
list strs = []

function index(str):     // turn str into an int starting from 0.
    if str not in strs:
        strs += str              // add str to strs
        return strs.lastIndex         // IE: strs.length - 1
    else:
        return index of str in strs       // (findIndex)



match
    ch = Channel

    sigs= (SendSignal or SignalReception)* (port:
        ch.port1         or ch.port2  or
        ch.sourcePort    or ch.targetPort
    )

replace with

    ch

    copy sigs into sigs._type* (signalName: sigs.signalName + '_' + index(ch.name))
```

***Code fragment 19:** The **Identify Channels** transformation in **SLCOtrans v2**.*

Finally, we emphasize that a match in this example contains one channel and (possibly) multiple signals.

Code fragment 20 shows the *Identify Channels* transformation in the GPTL-like syntax. For this transformation, SLCOtrans v2 is not much more or less concise than a traditional language.

```
list strs = []

function index(str):      // turn str into an int starting from 0.
    if str not in strs:
        strs += str              // add str to strs
        return strs.lastIndex         // IE: strs.length - 1
    else:
        return index of str in strs       // (findIndex)



for ch in Channel.all:

    for sigCmd in SendSignal.all + SignalReception.all:

        if  ch.port1         = sigCmd.port or
            ch.port2         = sigCmd.port or
            ch.sourcePort    = sigCmd.port or
            ch.targetPort    = sigCmd.port:

            sigCmd.signalName += '_' + index(ch.name)
```

***Code fragment 20:** The **Identify Channels** transformation in **pseudocode**.*

8.4 Making all Signal Names Equal

The **Names to Arguments** transformation changes the names of signals to a fixed name ('signal') and supplies the original name as an argument.

Code fragment 21 shows the implementation in SLCOtrans v2. The only new feature is the double quotes in `"Signal"`. This represents the literal string "Signal". **Code fragment 21** demonstrates that the creation of objects can be nested arbitrarily. Separate cases are needed for `SendSignal` and `SignalReception`, because the `arguments` attribute in the abstract model slightly differs between them. It is also needed to add an additional string parameter to channels, to send the old name. This is done in the last block.

```
match
    snd = SendSignal

replace with
    copy snd into SendSignal (
        signalName:  "Signal"
        arguments:   StringConstantExpression(value: snd.signalName) + snd.arguments  //sendSig has expr.
    )



match
    rcv = SignalReception

replace with
    copy rcv into SignalReception (
        signalName:  "Signal"
        arguments:   SignalArgumentExpression(            // sigRec has SigArg.
            expression: StringConstantExpression(value: rcv.signalName)
        ) + rcv.arguments
    )



match
    ch = Channel

replace with
    copy ch into Channel (
        argumentTypes: ArgumentType(type: PrimitiveType.String) + ch.argumentTypes
    )
```

***Code fragment 21:** The **Names to Arguments** transformation in **SLCOtrans v2**.*

Code fragment 22 shows the transformation in the GPTL-like syntax. For this transformation, SLCOtrans v2 is not much more or less concise than a traditional language.

```
for snd in SendSignal.all:

    snd.signalName   = "Signal"

    prepend StringConstantExpression(value: snd.signalName) to snd.arguments    // sendSig has expr.


for rcv in SignalReception.all:

    rcv.signalName   = "Signal"

    prepend SignalArgumentExpression(                      // sigRec has SigArg.
        expression: StringConstantExpression(value: rcv.signalName)
    ) to rcv.arguments


for ch in Channel.all:

    prepend ArgumentType(type: PrimitiveType.String) to ch.argumentTypes
```

***Code fragment 22:** The **Names to Arguments** transformation in **pseudocode**.*

8.5 Removing Unused Classes

The **Remove Unused Classes** transformation "removes all uninstantiated classes from a model." [13]. This is an auxiliary transformation.

Code fragment 23 shows the implementation in SLCOtrans v2. The `not` keyword is new. The 'match'-block indicates that a match is returned when there is a `Class`, for which there is no `Object` with that `Class` as its attribute '`class`'.

Code fragment 24 shows the transformation in the GPTL-like syntax. For this transformation, being based on pattern matching makes SLCOtrans v2 slightly more concise and natural than a traditional language.

```
match
    cls = Class
    not Object(class: cls)

replace with
    // nothing
```

***Code fragment 23:** The **Remove Unused Classes** transformation in **SLCOtrans v2**.*

```
for cls in Class.all:

    if not Object.all.exists(obj|obj.class = cls):
        delete class
```

***Code fragment 24:** The **Remove Unused Classes** transformation in **pseudocode**.*

8.6 Replacing a Bidirectional Channel by two Unidirectional Channels

The **Bidirectional To Unidirectional (Bi2Uni)** transformation converts communication over bidirectional channels to functionally equivalent communication over unidirectional channels.

It converts bidirectional channels into two unidirectional channels, one in each direction. The ports to which the bidirectional channels are connected are split into ports specific for each unidirectional channel.

Code fragment 25 shows the implementation in SLCOtrans v2. Because the unused ports need to be replaced only after the rest of the transformation has finished, we introduce the '`stage #`'-block. It enforces the order in which fragments are executed.

First the bidirectional channels and statements communicating signals over it are matched. Then the (bidirectional) channels are replaced by two unidirectional channels. Note that it is not necessary to use commas to separate attribute overrides. The unidirectional channels are connected to two new

ports, suffixed '`_send`' and '`_receive`'. The old ports are kept for now, and therefore not stored in a transformation variable stated at the start of a line in the 'match'-block, in case for example other unidirectional channels are connected to it. Also, the statements that communicate signals are modified to communicate over the new ports. Finally, the old ports that remain unused after the first stage are removed.

```
stage 1:
    match
        bich= BidirectionalChannel(      // autocast: typeless attr match. Overrides:
            object1:     obj1
            port1:       prt1
            object2:     obj2
            port2:       prt2
        )

        prt1snds = SendSignal*        (port: prt1) // contains sigNm, args
        prt1rcvs = SignalReception*   (port: prt1) // contains sigNm, args, cond
        prt2snds = SendSignal*        (port: prt2) // contains sigNm, args
        prt2rcvs = SignalReception*   (port: prt2) // contains sigNm, args, cond

    replace with
        copy bich into UnidirectionalChannel( // autocast: typeless attr match. Overrides:
            sourceObject:    obj1
            sourcePort:      sndPrt1
            targetObject:    obj2
            targetPort:      rcvPrt2
        )
        copy bich into UnidirectionalChannel( // autocast: typeless attr match. Overrides:
            sourceObject:    obj2
            sourcePort:      sndPrt2
            targetObject:    obj1
            targetPort:      rcvPrt1
        )


        // cannot simply rename ports, because 2 new for 1 old.

        // included here because only ports connected to a bich should be replaced:

        sndPrt1  = copy prt1 into Port(
            name: prt1.name + '_send'
        )

        rcvPrt1  = copy prt1 into Port(
            name: prt1.name + '_receive'
        )

        sndPrt2  = copy prt2 into Port(
            name: prt2.name + '_send'
        )

        rcvPrt2  = copy prt2 into Port(
            name: prt2.name + '_receive'
        )


        // included here because only signals connected to ports connected to a bich should be replaced:

        copy prt1snds into SendSignal*        (port: sndPrt1)
        copy prt1rcvs into SignalReception*   (port: rcvPrt1)
        copy prt2snds into SendSignal*        (port: sndPrt2)
        copy prt2rcvs into SignalReception*   (port: rcvPrt2)

stage 2:
    match
        prt = Port
        not UnidirectionalChannel(sourcePort or targetPort: prt)
        not BidirectionalChannel  (port1 or port2: prt)

    replace with
        // nothing
```

***Code fragment 25:** The **Bi2Uni** transformation in **SLCOtrans v2**.*

The operation '`copy xx into yy`' is not only used in **Code fragment 25** to copy all attributes of `xx` into all attributes of `yy`, if possible for that attribute, but it is also used to trace which match is used for which replacement or addition. Without this, it would be necessary to indicate for example in which class the new ports should be added. But now, the ports are automatically added to the same class as the one from which they are derived (unless this behavior is overwritten by other code).

Code fragment 26 shows the transformation in a traditional kind of model manipulation language, not based on pattern matching. The first part replaces the ports. The second part replaces the bidirectional channels. And the last part modifies the signal communication statements. The removal of unused ports is left out.

```
// model = Slco!Model.all.first;  // <- needs to be used in EOL, because there is no other way to get
                                  // the (single) model object through which everything in the model can
                                  // be found.

For bich in BidirectionalChannel.all:

    // cannot simply rename ports, because 2 new for 1 old.

    sndPrt1 = Port(
        name: bich.port1.name + '_send'
    )

    rcvPrt1 = Port(
        name: bich.port1.name + '_receive'
    )

    sndPrt2 = Port(
        name: bich.port2.name + '_send'
    )

    rcvPrt2 = Port(
        name: bich.port2.name + '_receive'
    )

    add sndPrt1, rcvPrt1
    to  object1.class.ports

    add sndPrt2, rcvPrt2
    to  object2.class.ports


    // + keep a list of old ports and remove the unused ones after the loop.


    uni1 = bich copy into UnidirectionalChannel(  // autocast: typeless attr match. overrides:
        sourceObject:    bich.object1
        sourcePort:      sndPrt1
        targetObject:    bich.object2
        targetPort:      rcvPrt2
    )

    uni2 = bich copy into UnidirectionalChannel(  // autocast: typeless attr match. overrides:
        sourceObject:    bich.object2
        sourcePort:      sndPrt2
        targetObject:    bich.object1
        targetPort:      rcvPrt1
    )

    replace  bich
    in  model.channels
    by  uni1, uni2


    for SendSignal snd in object1.class.stateMachines.transitions.statements:
        snd.port = sndPrt1

    for SignalReception rcv in object1.class.stateMachines.transitions.statements:
        rcv.port = rcvPrt1

    for SendSignal snd in object2.class.stateMachines.transitions.statements:
        snd.port = sndPrt2

    for SignalReception rcv in object2.class.stateMachines.transitions.statements:
        rcv.port = rcvPrt2
```

Code fragment 26: *The* ***Bi2Uni*** *transformation in* ***pseudocode****.*

The fragment …

```
“  for SendSignal snd in object1.class.stateMachines.transitions.statements:
       snd.port = sndPrt1                                                    ”
```

… is shorthand for: …

```
“  for sm in object1.class.stateMachines
       for transit in sm.transitions
           for SendSignal snd in transit.statements:
               snd.port = sndPrt1                    ”.
```

Though the attribute in (the sole instance of) object `Model` in the abstract model where the channels are stored is always the same, it is explicitly indicated in **Code fragment 26** that the (unidirectional) channels with which the bidirectional channels are replaced, need to be added in `model.channels` (`model` is the sole instance in this case).

In **Code fragment 25** in SLCOtrans v2 it was not necessary to do this, because matches are (by default) replaced at all places where they are referenced. The '`copy … into …`'-operation is used in SLCOtrans v2 to know which match is used for which replacement or addition. It is needed to know which match is this corresponding match, to know where references need to be modified for the replacement.

Even if a newly created channel is not derived from a match, it is still automatically added to `model.channels`. This feature is possible because SLCOtrans v2 can use knowledge about (the abstact model of) SLCO. In this sense, SLCOtrans v2 is a DSTL even though the abstract syntax is often used.

Note that the feature of using knowledge about SLCO to add elements in the right place is not possible for adding ports to classes, because it is necessary to know to which class it needs to be added. In that case, the '`copy … into …`'-operation is needed to trace the match.

The two features just described can make SLCOtrans v2 more concise than traditional model manipulation languages. In the case of the *Bi2Uni* transformation, only the first feature, of not needing to know the places where objects need to be replaced or added, causes SLCOtrans v2 to be slightly more concise than traditional model manipulation languages.

8.6.1 Looping templates

There is quite some repetition in **Code fragment 25**. We now introduce a way to *avoid writing repetitive code* in SLCOtrans v2. We call this technique *looping templates*. For this technique we extract the common part from patches of similar code and add combinations of arguments that we loop through into the places we indicate. This results in a shorter patch of code in which the repetitions are removed.

We will illustrate the technique using the example in **Code fragment 27**. This is the *Bi2Uni* transformation in **Code fragment 25**, but without removing unused ports. The looping templates are enclosed by `<!` and `!>` and are applied to the indented block after it. This indentation level is removed in the generated result.

First, we will give some small examples.

'`<! $1= 1/2/3: !>`' means:

'Repeat the block after this. In the first repetition, replace all occurrences of `$1` by '1' and in the second by '2' and in the third by '3'.'.

We will now show an arbitrary example (left is the same as right):

```
<! $1= 1/2/3: !>
    obj$1 = prt$1
```

```
obj1 = prt1
obj2 = prt2
obj3 = prt3
```

'`<!$1= (object, obj)/(port, prt): !>`' means:

'Repeat the block after this. In the first repetition, replace the first `$1` by 'object' and the second `$1` by 'obj' and the third `$1` by 'object' again, etc. In the first repetition, do the same for 'port' and 'prt'.'.

Used in an arbitrary example this is (left is the same as right):

```
<! $1= (object, obj)/(port, prt): !>
    example2.$1  = $1     // assign example2.$1
```

```
example2.object  = obj    // assign example2.object
example2.port    = prt    // assign example2.port
```

The tuples can of course be extended with more elements.

The usefulness especially appears when multiple iterators (`$1` etc) are used:

'`<! $1= object/port, $2= 1/2: !>`' means:

'Repeat the block after this for each value-combination of `$1` and `$2`. Replace `$1` by 'object' and 'port'. Replace `$2` by '1' and '2'.' When viewed as loops, `$1` would be the inner loop, and `$2` the outer loop. So '`<! $1= object/port, $2= 1/2: !>`' is equivalent to '`<! $2= 1/2: !><! $1= object/port: !>`'.

We will now show an (arbitrary) example again (left is the same as right):

```
<! $1= object/port,  $2= 1/2: !>
    $1$2     = example3.$1
```

```
object1  =  example3.object
object2  =  example3.object
port1    =  example3.port
port2    =  example3.port
```

Finally, we also present an option to choose a fixed element from a tuple. '`$1.1`' means 'Replace this by the first value in the `$1` -tuple.'

In **Code fragment 27**, inside the block after the looping template with `$3` in it, is another looping template. The iteration of the inner one is increased first. This was also indicated by the numbers of the iterators though.

Now we have presented all information needed to understand **Code fragment 27**. In **Code fragment 25** the equivalent is shown of **Code fragment 27**, when the 'stage 2'-block (i.e.,the part removing unused ports) and the first line are removed.

```
match
    bich= BidirectionalChannel(      // autocast: typeless attr match. overrides:
        <! $1= (object, obj)/(port, prt), $2= 1/2: !>
            $1$2:        $1$2
    )

        <! $1= (snd, SendSignal)/(rcv, SignalReception), $2= 1/2: !>
            prt$2$1s = $1*        (port: prt$2)    // contains sigNm, args (, cond)

replace with
    <! $3= (1,2)/(2,1): !>
        copy bich into UnidirectionalChannel(     // autocast: typeless attr match. overrides:
            <! $1= (Object, obj)/(Port, $2Prt), $2= (snd/rcv, source/target, $3/$3): !>
                $2$1:    $1$2
        )


    // cannot simply rename ports, because 2 new for 1 old.

    // included here because only ports connected to a bich should be replaced:

    <! $1= (snd, send)/(rcv, receive),    $2= 1/2: !>
        $1Prt$2  = copy prt$2 into Port(
            name: prt$2.name + '_$1'
        )

    // included here because only signals connected to ports connected to a bich should be replaced:

    <! $1= (snd, SendSignal)/(rcv, SignalReception),   $2= 1/2: !>
        copy prt$2$1.1s into $1.2*        (port: $1.1Prt$2)
```

Code fragment 27: *The **Bi2Uni** transformation (without removing unused ports) in **SLCOtrans v2** using **looping templates**.*

Looping templates are a nice way to avoid writing repetitive code, but we will avoid them in the implementation of other transformations, to keep the code as accessible to readers as possible and to keep the comparisons fair.

8.7 Cloning Classes

CloneClasses is an auxiliary transformation. We implemented two variants of this transformation. The first variant splits all classes used by multiple objects into copies. The second variant takes a channel as input and creates copies of the classes of the objects connected to that channel.

The first variant is shown in **Code fragment 28**. '`deep copy … into …`' means the same as '`copy … into …`', but a deep copy is made. Normally, attributes in a copy refer to the same objects as the original. In a deep copy, attributes refer to (deep) copies of the object originally referred to.

Code fragment 28 also demonstrates the copying of a single `Class` into multiple `Class`es. Previously we have only shown the copying of a *list of multiple* objects into multiple objects.

`xx_loopCount` is a built-in operation. It returns the current iteration number while iterating though a list. It is inspired by EOL. `xx` indicates the 'loop count' of which iterator is used.

A list followed by a dot followed by an expression that results in an integer (such as '`cls_cps.(objs._loopCount-1)`') indicates an array index (starting from 0). So, '`cls_cps.3`' would refer to the 4th `Class` in `cls_cps`.

In **Code fragment 28**, the name of each copy is suffixed with '_c' followed by a unique number. We also suffixed the name of the objects to indicate of which objects the class has been copied and which copy they use. This is done in the commented out section, because it is not imperative to the transformation.

Note that in `Object*(class: cls)`, there is an asterisk after '`Object`', but not after '`cls`'. This means that all objects with the same class are matched.

```
match
    objs = Object*(class: cls)

replace with

    cls_cps = deep copy cls into Class*(name: cls.name + "_c" + objs._loopCount)

    copy objs into Object*(
        /*name: objs.name + "_c" + objs._loopCount */
        class: cls_cps.(objs._loopCount-1)
    )
```

Code fragment 28: *The first variant of the* ***CloneClasses*** *transformation in* ***SLCOtrans v2****.*

Code fragment 29 shows the first variant of the transformation in a traditional kind of model manipulation language, not based on pattern matching. Note that **Code fragment 28** in SLCOtrans v2 suffixes with sequential numbers per class, while **Code fragment 29** in pseudocode does not. **Code fragment 29** is easier to understand than **Code fragment 28**.

```
for cls in Class.all:

    for obj in Object.all:

        if obj.class = cls:

            obj.class = deep copy cls into Class(name: cls.name + "_c" + obj._loopCount)
            /*obj.name += "_c" + obj._loopCount */
```

Code fragment 29: *The first variant of the* ***CloneClasses*** *transformation in* ***pseudocode****.*

Code fragment 30 shows the second variant of the transformation. It takes a channel as input and creates (deep) copies of the classes of the objects connected to that channel. The classes are replaced by the copies. Their name is suffixed with '_c'. Optionally and commented out, the names of the objects and channels for which this occurs can also be suffixed with '_c'.

We prefixed the channel argument '`ch`' used as input with '`$`', but this is really not necessary. Any transformation variable could be supplied as an argument and this would become an additional restriction to match with.

```
// arg $ch           // class-split objs connected to arg $ch

match
    $ch = UnidirectionalChannel(
        sourceObject: obj1
        targetObject: obj2
    ) or
    BidirectionalChannel(
        object1: obj1
        object2: obj2
    )

    obj1
    obj2

replace with

    $ch
    // copy $ch into $ch._type(name: $ch.name + "_c")  // REPLACE above $ch with this for channel "_c".

    cls1_cp = deep copy obj1.class into Class(name: obj1.class.name + "_c")
    cls2_cp = deep copy obj2.class into Class(name: obj2.class.name + "_c")

    copy obj1 into Object(/*name: obj1.name + "_c",*/ class: cls1_cp)
    copy obj2 into Object(/*name: obj2.name + "_c",*/ class: cls2_cp)
```

Code fragment 30: *The second variant of the* ***CloneClasses*** *transformation in* ***SLCOtrans v2****.*

Code fragment 31 shows the second variant of the transformation in a traditional kind of model manipulation language, not based on pattern matching. It is more concise than **Code fragment 30**. Note that the 'match'-block in **Code fragment 30** is spread out over multiple lines for clarity, but this is not required.

```
// arg $ch           // class-split objs connected to arg $ch


/*$ch.name += "_c"*/

for obj in $ch.sourceObject, $ch.targetObject, $ch.object1, $ch.object2

    obj.class = deep copy obj.class into Class(name: obj.class.name + "_c")
    /*obj.name += "_c"*/
```

Code fragment 31: *The second variant of the* ***CloneClasses*** *transformation in* ***pseudocode****.*

8.8 Reducing the Number of Channels

The **MergeChannels** transformation merges multiple channels into one. That channels have to be between the same two objects, have the same communication type and direction, and support the same argument types.

Code fragment 32 shows the implementation in SLCOtrans v2. '`counter++`' means: 'increase `counter` by 1'. In the fragment '`unichs = UnidirectionalChannel*( … )`' each attribute has to have the same value between matched channels, except those where the transformation variable is followed by an asterisk. '`srcPrts*`' and '`tgtPrts*`' are stated at the beginning of al line in the match statement. This needs to be done so the old ports are removed. The ports in signal communication statements are automatically changed to the new ports, because SLCOtrans v2 replaces matches in all places they are referenced. This is the second feature mentioned in the discussion in **Section 8.6: 'Replacing a Bidirectional Channel by two Unidirectional Channels'**.

Virtually the same as for unidirectional channels is done for bidirectional channels in the second 'match/replace'-block. In the implementation we provide, bidirectional channels are only joined if the `object1` attribute of one channel is equal to the `object1` attribute of another channel, etc. The

transformation could be extended by also joining when the `object1` attribute of one (bidirectional) channel is equal to the `object2` attribute of another (bidirectional) channel, etc.

```
counter = 1

function count():
    counter++
    return counter


match
    unichs = UnidirectionalChannel*(
        channelType:     chType
        argumentTypes:   argTypes
        sourceObject:    srcObj
        sourcePort:      srcPrts*
        targetObject:    tgtObj
        targetPort:      tgtPrts*
    )

    srcPrts*         // necessary for merge/rm
    tgtPrts*

replace with
    copy unichs into UnidirectionalChannel(
        name:        'mergedChannel' + count()
        sourcePort:  newSrcPrt
        targetPort:  newTgtPrt
    )

    newSrcPrt = copy srcPrts* into Port(name: 'mergedPort')
    newTgtPrt = copy tgtPrts* into Port(name: 'mergedPort')

    // SendSignal.port and SignalReception.port solved automatically.
    //(different from Bi2Uni b/c there it cant be deduced what to replace by.)



match
    bichs = BidirectionalChannel*(
        channelType:     chType
        argumentTypes:   argTypes
        object1:         obj1         // NOT IMPLEMENTED: also if obj1 and obj2 are reversed, (for bich).
        port1:           prt1s*
        object2:         obj2
        port2:           prt2s*
    )

    prt1s*           // necessary for merge/rm
    prt2s*

replace with
    copy bichs into BidirectionalChannel(
        name:    'mergedChannel' + count()
        port1:   newPrt1
        port2:   newPrt2
    )

    newPrt1 = copy prt1s* into Port(name: 'mergedPort')
    newPrt2 = copy prt2s* into Port(name: 'mergedPort')

    // SendSignal.port and SignalReception.port solved automatically.
    //(different from Bi2Uni b/c there it cant be deduced what to replace by.)
```

***Code fragment 32:** The **MergeChannels** transformation in **SLCOtrans v2**.*

Code fragment 33 shows the transformation in a traditional kind of model manipulation language, not based on pattern matching. For the *MergeChannels* transformation, SLCOtrans v2 provides benefits over traditional languages by not having to explicitly replace the ports in all places they occur, such as in the signal communication statements.

```
counter = 1

fn count():
    counter++
    return counter

for unich1 in UnidirectionalChannel.all:

    for unich2 in UnidirectionalChannel.all:

        if  unich1.channelType       = unich2.channelType and
            unich1.argumentTypes     = unich2.argumentTypes   and
            unich1.sourceObject      = unich2.sourceObjectand
            unich1.targetObject      = unich2.targetObject:

            unich1.name              = 'mergedChannel' + count()
            unich1.sourcePort.name   = 'mergedPort'
            unich1.targetPort.name   = 'mergedPort'

            for snd in SendSignal.all:
                if snd.port = unich2.sourcePort:
                    snd.port = unich1.sourcePort

            for rcv in SignalReception.all:
                if rcv.port = unich2.targetPort:
                    rcv.port = unich1.targetPort

            delete unich2



for bich1 in BidirectionalChannel.all:

    for bich2 in BidirectionalChannel.all:

        if  bich1.channelType    = bich2.channelType      and
            bich1.argumentTypes  = bich2.argumentTypes    and
            bich1.object1        = bich2.object1          and // NOT IMPL: also if obj1 & obj2 reversed.
            bich1.object2        = bich2.object2:

            bich1.name           = 'mergedChannel' + count()
            bich1.port1.name     = 'mergedPort'
            bich1.port2.name     = 'mergedPort'

            for snd in SendSignal.all:
                if snd.port = bich2.port1:    // NOT IMPLEMENTED: also if port1 and port2 are reversed.
                    snd.port = bich1.port1

            for rcv in SignalReception.all:
                if rcv.port = bich2.port2:
                    rcv.port = bich1.port2

            delete bich2
```

Code fragment 33: *The* ***MergeChannels*** *transformation in* ***pseudocode****.*

8.9 Lossless Communication over a Lossy Channel

The **Lossless2Lossy** transformation transforms lossless channels to lossy channels over which lossless communication is performed. This is done by adding four auxiliary objects to implement the Concurrent Alternating Bit Protocol (CABP). An overview is shown in **Figure 7**. One object is responsible for sending, one for receiving, one for acknowledging, and one for receiving acknowledgements.

The Sender has 3 state machines: one for receiving from the original source object, one for sending to the Receiver, and one for receiving from the Acknowledgement Receiver.

The Receiver also has 3 state machines: one for sending to the original target object, one for receiving from the Sender, and one for sending to the Acknowledgement Sender.

The Acknowledgement Sender has 2 state machines: one for receiving from the Receiver, and one for sending acknowledgements to the Acknowledgement Receiver.

The Acknowledgement Receiver has 2 state machines: one for sending to the Sender, and one for receiving acknowledgements from the Acknowledgement Sender.

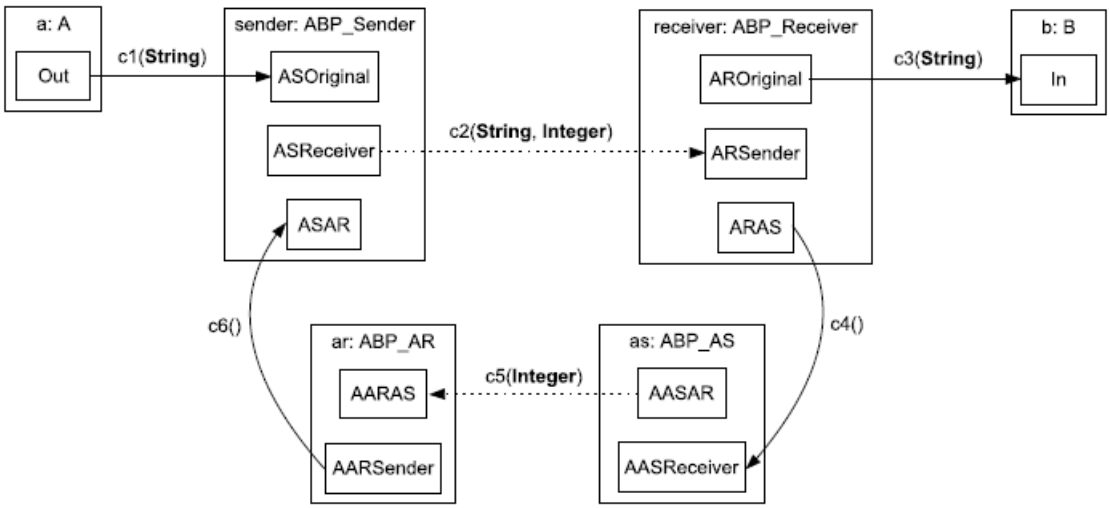

Figure 7: *Overview of the Concurrent Alternating Bit Protocol (**CABP**) in the graphical notation of SLCO.*

Code fragment 34 shows the implementation in SLCOtrans v2. '`[ArgumentType.String]`' is a list with one element in it. Between triple quotes fragments of SLCO are included. Inside these fragments of SLCO, fragments of SLCOtrans can be used between double braces. One particular detail is that the name of the signal sent from the original source and to the original target is fixed to 'signal'.

We have not implemented this transformation in pseudocode, because this would take too much time. The *Lossless2Lossy* transformation is an excellent example of a case where DSTLs provide significant advantages. It would take significantly more effort to write this transformation in a traditional model manipulation language. Being able to use a large part of the desired output in SLCO in the SLCOtrans v2 implementation makes it significantly easier to write this transformation.

```
match
    ch = UnidirectionalChannel(
        channelType:     ChannelType.AsynchronousLossless
        argumentTypes:   [ArgumentType.String]
        sourceObject:    obj1
        targetObject:    obj2
    )

    snds = SendSignal*(
        name: 'Signal'
        port: ch.sourcePort
    )

    rcvs = SignalReception*(
        name: 'Signal'
        port: ch.targetPort
    )

replace with

    snds             // snds.port.name = Out
    rcvs             // rcvs.port.name = In

    '''
    classes
        ABP_Sender {
            ports
                ASOriginal
                ASReceiver
                ASAR

            state machines
                Sender {
                    variables
                        Integer s = 0
                        String d

                    initial
                        Zero

                    state
                        One

                    transitions
                        ZeroToOne from Zero to One {
                            receive Signal(d) from ASOriginal
                        }

                        OneToZero from One to Zero {
                            receive Acknowledge() from ASAR;
                            s := (1 - s)
                        }

                        SenderOneToOne from One to One {
                            send Message(d, s) to ASReceiver
                        }
                }
        }

        ABP_AR {
            ports
                AARAS
                AARSender

            state machines
                AR {
                    variables
                        Integer b = 0

                    initial
                        Zero

                    transitions
                        ZeroToZeroAck from Zero to Zero {
                            receive Acknowledge([[b]]) from AARAS;
                            send Acknowledge() to AARSender;
                            b := (1 - b)
                        }

                        ZeroToZero from Zero to Zero {
                            receive Acknowledge([[(1 - b)]]) from AARAS
                        }
                }
        }

        ABP_Receiver {
            ports
                AROriginal
```

```
                ARSender
                ARAS

            state machines
                Receiver {
                    variables
                        Integer r = 0
                        String d

                    initial
                        Zero

                    transitions
                        ZeroToZeroAck from Zero to Zero {
                            receive Message(d, [[r]]) from ARSender;
                            send Signal(d) to AROriginal;
                            send Acknowledge() to ARAS;
                            r := (1 - r)
                        }

                        ZeroToZero from Zero to Zero {
                            receive Message(d, [[(1 - r)]]) from ARSender
                        }
                }
        }

        ABP_AS {
            ports
                AASAR
                AASReceiver

            state machines
                AS {
                    variables
                        Integer b = 1

                    initial
                        Zero

                    transitions
                        ZeroToZeroAck from Zero to Zero {
                            receive Acknowledge() from AASReceiver;
                            b := (1 - b)
                        }

                        ASZeroToZero from Zero to Zero {
                            send Acknowledge(b) to AASAR
                        }
                }
        }

    objects
        {{obj1.name}}_OutABP_Sender: ABP_Sender
        {{obj1.name}}_OutABP_AR: ABP_AR
        {{obj2.name}}_InABP_Receiver: ABP_Receiver
        {{obj2.name}}_InABP_AS: ABP_AS

    channels
        {{obj1.name}}_Out_Original_to_Sender(String) sync
            from {{obj1.name}}.{{ch.sourcePort.name}} to {{obj1.name}}_OutABP_Sender.ASOriginal
        {{obj1.name}}_Out_AR_to_Sender() sync
            from {{obj1.name}}_OutABP_AR.AARSender to {{obj1.name}}_OutABP_Sender.ASAR
        {{obj2.name}}_In_Receiver_to_Original(String) sync
            from {{obj2.name}}_InABP_Receiver.AROriginal to {{obj2.name}}.{{ch.targetPort.name}}
        {{obj2.name}}_In_Receiver_to_AS() sync
            from {{obj2.name}}_InABP_Receiver.ARAS to {{obj2.name}}_InABP_AS.AASReceiver
        {{obj1.name}}_Out_Sender_to_Receiver(String, Integer) async lossy
            from {{obj1.name}}_OutABP_Sender.ASReceiver to {{obj2.name}}_InABP_Receiver.ARSender
        {{obj1.name}}_Out_AS_to_AR(Integer) async lossy
            from {{obj2.name}}_InABP_AS.AASAR to {{obj1.name}}_OutABP_AR.AARAS
    '''
```

Code fragment 34: *The* ***MergeChannels*** *transformation in* ***SLCOtrans v2****.*

8.10 Synchronized Communication over Asynchronous Channels

The **Sync2Async** transformation converts a synchronous channel to an asynchronous channel, and the communication over it such that it is functionally equivalent. There are two variants of this transformation: a general variant and a simple variant. We only implemented the simple variant.

The simple variant is less complex, but it only works correctly on restricted cases. It “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.” [13].

```
// '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.'

function signalArgumentExpressions(exprs):

    sigArgExprs = []

    for expr in exprs:

        sigArgExprs += SignalArgumentExpression(expression: expr)

    return sigArgExprs


function expressions(sigArgs):

    exprs = []

    for sigArg in sigArgs:

        if sigArg._type = SignalArgumentVariable:

            exprs += VariableExpression(variable: sigArg.variable)

        else:   // SignalArgumentExpression

            exprs += sigArg.expression

    return exprs


match
    ch = BidirectionalChannel(channelType: ChannelType.Synchronous)

    snd = SendSignal(port: ch.port1 or ch.port2)

    rcv = SignalReception(port: ch.port1 or ch.port2)

replace with
    copy ch into BidirectionalChannel(channelType: ChannelType.AsynchronousLossless)

    copy snd into SendSignal(signalName: 'Send_' + snd.signalName)
    SignalReception  (
        signalName: 'Acknowledge_'    + snd.signalName
        arguments:   signalArgumentExpressions(snd.arguments)
    )

    copy rcv into SignalReception(signalName: 'Send_' + rcv.signalName)
    SendSignal (
        signalName: 'Acknowledge_'    + rcv.signalName
        arguments:   expressions(rcv.arguments)
    )


match
    ch = UnidirectionalChannel(channelType: ChannelType.Synchronous)

    snd = SendSignal(port: ch.sourcePort)

    rcv = SignalReception(port: ch.targetPort)

replace with
    copy ch into BidirectionalChannel(
        channelType:     ChannelType.AsynchronousLossless
        port1:           ch.sourcePort
        port2:           ch.targetPort
    )

    copy snd into SendSignal(signalName: 'Send_' + snd.signalName)
    copy snd into SignalReception (
        signalName: 'Acknowledge_'    + snd.signalName
        arguments:   signalArgumentExpressions(snd.arguments)
    )

    copy rcv into SignalReception(signalName: 'Send_' + rcv.signalName)
    copy rcv into SendSignal (
        signalName: 'Acknowledge_'    + rcv.signalName
        arguments:   expressions(rcv.arguments)
    )
```

Code fragment 35: *The* ***Sync2Async*** *transformation in* ***SLCOtrans v2****.*

After the transformation, the receiving object sends acknowledgements and the sending objects waits until it receives them. Unidirectional channels are also transformed into bidirectional channels,

because otherwise the acknowledgements cannot be sent over them. Our implementation is shown in **Code fragment 35**.

The implementation matches channels and the signal communication statements that communicate over them. A signal reception or a signal send statement is added for the acknowledgement after each opposite statement. The functions are used to package the arguments, because there is a slight difference between the way arguments are represented in `SendSignal`s and `SignalReception`s in the metamodel of SLCO.

Code fragment 36 shows the *Sync2Async* transformation in a traditional kind of model manipulation language, not based on pattern matching. For this transformation, SLCOtrans v2 is slightly more concise than a traditional language. In **Code fragment 36**, channels need to be explicitly added to `model.channels` and signal communication statements need to be explicitly added to the `statements` attribute of a `Transition`. This does not need to be done in SLCOtrans v2.

```
// model = Slco!Model.all.first;  // <- needs to be used in EOL, because there is no other way to get
                                  // the (single) model object through which everything in the model can
                                  // be found.

for unich in UnidirectionalChannel.all:

    bich = unich -> BidirectionalChannel(
        channelType: ChannelType.AsynchronousLossless
        port1:           unich.sourcePort
        port2:           unich.targetPort
    )

    add bich
    to  model.channels

    for transit in Transition.all:

        for SendSignal snd in transit.statements:

            if snd.port = unich.sourcePort:

                snd.signalName = 'Send_' + snd.signalName

                argExprs = []

                for arg in snd.arguments:

                    argExprs += SignalArgumentExpression(expression: arg)

                insert  SignalReception(
                            signalName: 'Acknowledge_' + snd.signalName
                            arguments:  argExprs
                        )
                at      index after snd
                in      transit.statements

        for SignalReception rcv in transit.statements:

            if rcv.port = unich.targetPort:

                rcv.signalName = 'Send_' + rcv.signalName

                argExprs = []

                for arg in rcv.arguments:

                    if arg._type = SignalArgumentVariable:

                        argExprs += VariableExpression(variable: arg.variable)

                    else:   // SignalArgumentExpression

                        argExprs += arg.expression

                insert  SendSignal(
                            signalName: 'Acknowledge_' + rcv.signalName
                            arguments:  argExprs
                        )
                at      index after rcv
                in      transit.statements
```

```
for bichOld in BidirectionalChannel.all:

    for prtTuple in ((bichOld.port1, bichOld.port2), (bichOld.port2, bichOld.port1)):

        bichNew = bichOld -> BidirectionalChannel(
            channelType: ChannelType.AsynchronousLossless
            port1:           prtTuple.1
            port2:           prtTuple.2
        )

        add bichNew
        to  model.channels

        for transit in Transition.all:

            for SendSignal snd in transit.statements:

                if snd.port = prtTuple.1:

                    snd.signalName = 'Send_' + snd.signalName

                    argExprs = []

                    for arg in snd.arguments:

                        argExprs += SignalArgumentExpression(expression: arg)

                    insert  SignalReception(
                                signalName: 'Acknowledge_' + snd.signalName
                                arguments:  argExprs
                            )
                    at      index after snd
                    in      transit.statements

            for SignalReception rcv in transit.statements:

                if rcv.port = prtTuple.2:

                    rcv.signalName = 'Send_' + rcv.signalName

                    argExprs = []

                    for arg in rcv.arguments:

                        if arg._type = SignalArgumentVariable:

                            argExprs += VariableExpression(variable: arg.variable)

                        else:   // SignalArgumentExpression

                            argExprs += arg.expression

                    insert  SendSignal(
                                signalName: 'Acknowledge_' + rcv.signalName
                                arguments:  argExprs
                            )
                    at      index after rcv
                    in      transit.statements
```

Code fragment 36: *The* ***Sync2Async*** *transformation in* ***pseudocode****.*

8.11 Exclusive Channels for Pairs of State Machines

The **ExclusiveChannels** transformation splits channels between objects into separate channels for each pair of state machines in those objects that communicate with each other. The ports connected to the channels are also split into separate ports. An example of the transformation, performed on two objects, each with two state machines communicating with every state machine in the other object, is shown in **Figure 8**.

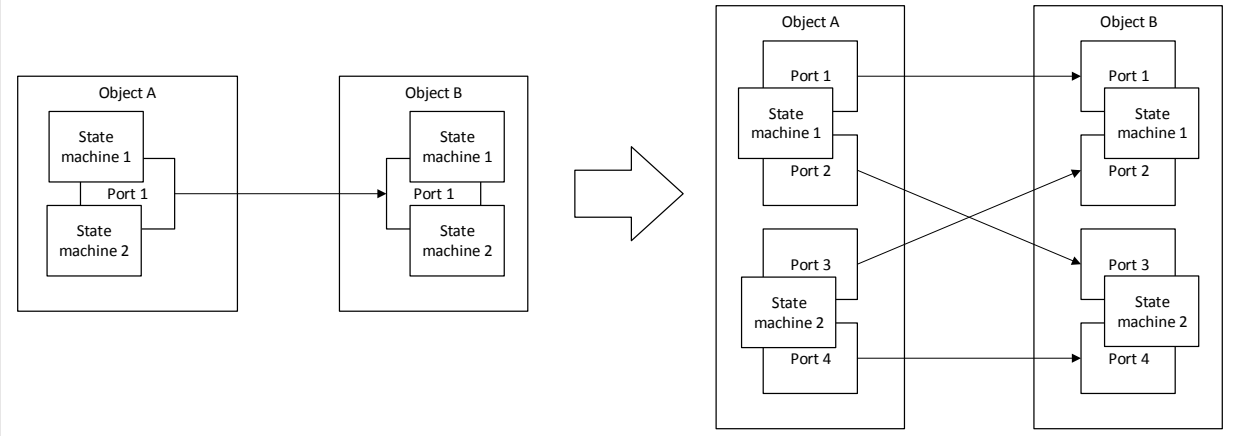

***Figure 8:** Overview of the **ExclusiveChannels** transformation performed on two objects, each with two state machines communicating with every state machine in the other object.*

Inside state machines, the transformation adds new states before and after signal communication statements, and separates the original transition in transitions for the part before and the part after the signal communication statement, and transitions with only a send signal statement to unique ports for each state machine the signal will reach, or vice versa for signal reception statements.

Code fragment 37 shows our implementation of a rather restricted variant of the transformation, in SLCOtrans v2. Our implementation is limited to two objects with two state machines each with one transition each. The transition in one state machine in the first object should contain an assignment, followed by a send signal statement, followed by another assignment. The transition in the other state machine in the first object should contain only a send signal statement. The same should hold for the second object, but with signal reception statements instead of send signal statements. It is also limited to unidirectional channels.

The implementation could be extended to include more cases, but doing so would quickly make the code much longer, take much more time to implement, and become much more difficult to implement and understand. It is not even certain that the full transformation can be specified at all in SLCOtrans v2, without adding large new features or modifications.

Some new syntax elements are used in **Code fragment 37**. '`[smA1, smA2]`' is a list comprehension containing two elements, specified on a single line, separated by a comma. '`ports: [ … ]`' shows a list comprehension containing four elements, specified on multiple lines. '`trA1 = Transition`' and '`A1ass1 = Assignment`' show that transformation variables can be assigned inside other statements. Finally, '`-`' is used to remove an element from a list.

```
match
    clsA = Class(
        stateMachines:   [smA1, smA2]
        ports:           [outPrt]
    )
    clsB = Class(
        stateMachines:   [smB1, smB2]
        ports:           [inPrt]
    )

    unich = UnidirectionalChannel(
        sourceObject:    Object(class: clsA)
        sourcePort:      outPrt
        targetObject:    Object(class: clsB)
        targetPort:      inPrt
    )


    smA1 = StateMachine(
        transitions: [
            trA1 = Transition(
                        statements: [A1ass1 = Assignment, sndA1 = SendSignal, A1ass2 = Assignment]
            )
        ]
    )

    smA2 = StateMachine(
        transitions: [
            trA2 = Transition(statements: [sndA2 = SendSignal])
        ]
    )

    smB1 = StateMachine(
        transitions: [
            trB1 = Transition(
                        statements: [B1ass1 = Assignment, rcvB1 = SignalReception, B1ass2 = Assignment]
            )
        ]
    )

    smB2 = StateMachine(
        transitions: [
            trB2 = Transition(statements: [rcvB2 = SignalReception])
        ]
    )


replace with
    copy clsA into Class(
        ports: [
            outPrtA1B1 = Port(name: outPrt.name + smA1 + '_' + smB1)
            outPrtA1B2 = Port(name: outPrt.name + smA1 + '_' + smB2)
            outPrtA2B1 = Port(name: outPrt.name + smA2 + '_' + smB1)
            outPrtA2B2 = Port(name: outPrt.name + smA2 + '_' + smB2)
        ]
    )
    copy clsB into Class(
        ports: [
            inPrtA1B1 = Port(name: inPrt.name + smA1 + '_' + smB1)
            inPrtA1B2 = Port(name: inPrt.name + smA1 + '_' + smB2)
            inPrtA2B1 = Port(name: inPrt.name + smA2 + '_' + smB1)
            inPrtA2B2 = Port(name: inPrt.name + smA2 + '_' + smB2)
        ]
    )



    copy smA1 into StateMachine(
        vertices: smA1.vertices
                    + A1_newState1 = Vertex(name: 'newState1')
                    + A1_newState2 = Vertex(name: 'newState2')
        transitions: [
            copy trA1 into Transition(target: A1_newState1, statements: [A1ass1])
            copy trA1 into Transition(
                                name: trA1.name + '_' + smB1.name
                                source: A1_newState1
                                target: A1_newState2
                                statements: [sndA1B1]
                            )
            copy trA1 into Transition(
                                name: trA1.name + '_' + smB2.name
                                source: A1_newState1
                                target: A1_newState2
                                statements: [sndA1B2]
                            )
            copy trA1 into Transition(source: A1_newState2, statements: [A1ass2])
        ]
    )
```

```
    sndA1B1 = copy sndA1 into SendSignal(port: outPrtA1B1)
    sndA1B2 = copy sndA1 into SendSignal(port: outPrtA1B2)


    copy smA2 into StateMachine(
        transitions: [
            copy trA2 into Transition(name: trA2.name + '_' + smB1.name, statements: [sndA2B1])
            copy trA2 into Transition(name: trA2.name + '_' + smB2.name, statements: [sndA2B2])
        ]
    )

    sndA2B1 = copy sndA2 into SendSignal(port: outPrtA2B1)
    sndA2B2 = copy sndA2 into SendSignal(port: outPrtA2B2)



    copy smB1 into StateMachine(
        vertices: smB1.vertices
                    + B1_newState1 = Vertex(name: 'newState1')
                    + B1_newState2 = Vertex(name: 'newState2')
        transitions: [
            copy trB1 into Transition(target: B1_newState1, statements: [B1ass1])
            copy trB1 into Transition(
                                name: trB1.name + '_' + smB1.name
                                source: B1_newState1
                                target: B1_newState2
                                statements: [rcvA1B1]
                            )
            copy trB1 into Transition(
                                name: trB1.name + '_' + smB2.name
                                source: B1_newState1
                                target: B1_newState2
                                statements: [rcvA1B2]
                            )
            copy trB1 into Transition(source: B1_newState2, statements: [B1ass2])
        ]
    )

    rcvA1B1 = copy rcvB1 into SignalReception(port: outPrtA1B1)
    rcvA1B2 = copy rcvB1 into SignalReception(port: outPrtA1B2)


    copy smB2 into StateMachine(
        transitions: [
            copy trB2 into Transition(name: trB2.name + '_' + smB1.name, statements: [rcvA2B1])
            copy trB2 into Transition(name: trB2.name + '_' + smB2.name, statements: [rcvA2B2])
        ]
    )

    rcvA2B1 = copy rcvB2 into SignalReception(port: outPrtA2B1)
    rcvA2B2 = copy rcvB2 into SignalReception(port: outPrtA2B2)



    copy unich into UnidirectionalChannel(
    // (unichA1B1 =)
        name:            outPrt.name + '_' + smA1.name + '_to_' + inPrt.name + '_' + smB1.name
        sourcePort:      outPrtA1B1
        targetPort:      inPrtA1B1
    )

    copy unich into UnidirectionalChannel(
        name:            outPrt.name + '_' + smA1.name + '_to_' + inPrt.name + '_' + smB2.name
        sourcePort:      outPrtA1B2
        targetPort:      inPrtA1B2
    )

    copy unich into UnidirectionalChannel(
        name:            outPrt.name + '_' + smA2.name + '_to_' + inPrt.name + '_' + smB1.name
        sourcePort:      outPrtA2B1
        targetPort:      inPrtA2B1
    )

    copy unich into UnidirectionalChannel(
        name:            outPrt.name + '_' + smA2.name + '_to_' + inPrt.name + '_' + smB2.name
        sourcePort:      outPrtA2B2
        targetPort:      inPrtA2B2
    )
```

Code fragment 37: *The* ***ExclusiveChannels*** *transformation in* ***SLCOtrans v2****.*

Code fragment 38 shows our implementation of the transformation in a traditional kind of model manipulation language, not based on pattern matching. This implementation is much less restricted than the implementation in SLCOtrans v2. It is still limited to unidirectional channels, but all other restrictions have been lifted. The implementation differs slightly from the way the transformation is defined though, in the sense that each statement is placed in a separate transition in this implementation. Though this is not how the transformation is defined, it is functionally equivalent.

Even though the implementation in SLCOtrans v2 is much less complete than the implementation in **Code fragment 38**, the implementation in **Code fragment 38** is already much more concise. We emphasize again that it is not even certain the full version of the transformation can be implemented at all in SLCOtrans v2, without significant modifications. The *ExclusiveChannels* transformation demonstrates that the limits of languages like SLCOtrans v2 are reached in more complicated cases. It becomes too difficult to write and the code becomes too long.

```
// this version splits transits at EVERY statement.

// (model)...        // in EOL the model should be assigned here.

newStateCounter = 0

for unich in UnidirectionalChannel.all:
    srcCls = unich.sourceObject.class
    tgtCls = unich.targetObject.class

    for srcSm in srcCls.stateMachines:
        if srcSm.transitions.statements.exists(SendSignal snd|snd.port = unich.sourcePort):

            for tgtSm in tgtCls.stateMachines:
                if tgtSm.transitions.statements.exists(SignalReception rcv|rcv.port = unich.targetPort):

                    srcPrt = Port(name: unich.sourcePort.name + '_' + srcSm.name + '_' + tgtSm.name)
                    srcCls.ports += srcPrt

                    tgtPrt = Port(name: unich.targetPort.name + '_' + srcSm.name + '_' + tgtSm.name)
                    tgtCls.ports += tgtPrt

                    model.channels += unich->UnidirectionalChannel(
                        name:       unich.sourcePort.name + '_' + srcSm.name + '_to_' +
                                    unich.targetPort.name + '_' + tgtSm.name
                        sourcePort: srcPrt
                        targetPort: tgtPrt
                    )

                    for transit in srcSm.transitions._old:
                                                    // how srcSm.transitions looked before the start.

                        subTransits = []

                        for stmt in transit.statements:

                            if stmt = transit.statements.first:
                                sourceState = transit.source
                            else:
                                leftState = Vertex(name: 'newState' + newStateCounter++)
                                srcSm.vertices += sourceState

                            if snd = transit.statements.last:
                                targetState = transit.target
                            else:
                                targetState = Vertex(name: 'newState' + newStateCounter++)
                                srcSm.vertices += targetState

                            newTransit = transit->Transition(
                                            name:       sourceState.name + '_to_' + targetState.name
                                            source:     sourceState
                                            target:     targetState
                                            statements: [stmt]
                                        )
                            srcSm.transitions += newTransit
                            subTransits += newTransit
```

```
                        for subTransit in subTransits:

                            for SendSignal snd in subTransit.statements:              // only one...

                                newTransit = transit->Transition(name: transit + '_' + tgtSm.name)
                                srcSm.transitions += newTransit
                                replace snd in newTransit by snd->SendSignal(port: srcPrt)

                                delete transit if it has not yet been deleted.


                    for transit in tgtSm.transitions._old:
                                                  // how tgtSm.transitions looked before the start.

                        subTransits = []

                        for stmt in transit.statements:

                            if stmt = transit.statements.first:
                                sourceState = transit.source
                            else:
                                leftState = Vertex(name: 'newState' + newStateCounter++)
                                srcSm.vertices += sourceState

                            if snd = transit.statements.last:
                                targetState = transit.target
                            else:
                                targetState = Vertex(name: 'newState' + newStateCounter++)
                                srcSm.vertices += targetState

                            newTransit = transit->Transition(
                                            name:        sourceState.name + '_to_' + targetState.name
                                            source:      sourceState
                                            target:      targetState
                                            statements:  [stmt]
                                        )
                            tgtSm.transitions += newTransit
                            subTransits += newTransit

                        for subTransit in subTransits:

                            for SignalReception rcv in subTransit.statements:         // only one...

                                newTransit = transit->Transition(name: transit + '_' + srcSm.name)
                                tgtSm.transitions += newTransit
                                replace rcv in newTransit by rcv->SignalReception(port: tgtPrt)

                                delete transit if it has not yet been deleted.

    delete unich.sourcePort
    delete unich.targetPort

    delete unich
```

Code fragment 38: *The* ***ExclusiveChannels*** *transformation in* ***pseudocode****.*

8.12 Reducing the Number of Objects

The **MergeObjects** transformation merges multiple objects into one object. The merged object contains the variables, ports, and state machines of all the original objects. Communication of the original object with each other over channels is replaced by communication using shared variables in the merged object. The transformation is only applicable under the condition that state machines communicate over unique unidirectional, synchronous channels.

Figure 9 shows an example of the protocol used to communicate using shared variables. On the left side are the state machines before the transformation, communicating over a channel, and on the right side are the state machines after the transformation, communicating using shared variables.

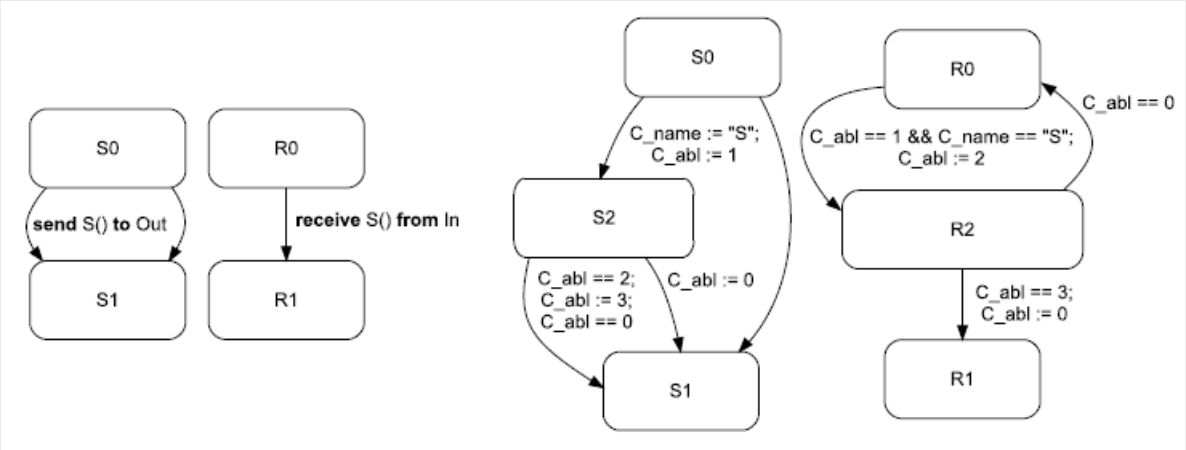

***Figure 9:** Communicating using **shared variables**. The **left** state machines communicate over a **channel**, **before** the transformation. The **right** state machines communicate using **shared variables**, **after** the transformation.*

Code fragment 39 shows the implementation of the *MergeObjects* transformation in SLCOtrans v2. Some new syntax elements are used in it. '`SendSignal+`' means 'match 1 or more `SendSignal`s'. '`snds = SendSignal+ in sndTransitsA.statements`' means that the `SendSignal`s matched here (stored in `snds`), should be in `sndTransitsA.statements`.

'`clsA.stateMachines.transitions`' means 'every `Transition` in `transitions` attribute in every state machine in the `stateMachines` attribute of `clsA`'.

'`copy (objA, objB) into Object`' means 'copy every attribute which is equal in `objA` and `objB` in the attribute with the same name in a new `Object`'.

'`replace each tr from sndTransitsA in xx by yy`' means 'replace every occurrence of an element `tr` from the list `sndTransitsA` in the list `xx` by `yy`'.

'`this`' refers to the (innermost) object being currently created. So in '`this._old.transitions`' in **Code fragment 39**, it refers to the `StateMachine` being currently created, so the fragment in **Code fragment 39** that determines this is '`StateMachine*()`'.

'`xx._old`' is a built-in operation that returns `xx` as it looked before the transformation started.

'`xx.collect(SendSignal snd)`' returns all elements of `xx` of type `SendSignal`. This operation also exists in EOL.

'`xx._count`' is a built-in operation that returns the number of elements in `xx`.

'`>`' is simply a 'larger than'-comparison.

```
// 'each pair of state machines that are part of two communicating objects must communicate over a
// unique unidirectional, synchronous channel.'

match
    unich = UnidirectionalChannel(
        sourceObject: objA
        targetObject: objB
    )

    objA = Object(class: clsA)
    objB = Object(class: clsB)

    clsA
    clsB

    sndTransitsA = Transition* in clsA.stateMachines.transitions
    rcvTransitsB = Transition* in clsB.stateMachines.transitions

    snds = SendSignal+       in sndTransitsA.statements    // '(port: unich.sourcePort)' by definition.
    rcvs = SignalReception+  in rcvTransitsB.statements    // '(port: unich.targetPort)' by definition.


replace with

    copy (objA, objB) into Object(name: objA.name + '_' + objB.name, class: clsAB)

    clsAB = copy (clsA, clsB) into Class(
        name:           clsA.name    + '_'    + clsB.name
        variables:      clsA.variables       + clsB.variables
                        + ablVar = Variable(
                            type:           PrimitiveType.Integer
                            name:           unich.name + '_abl'
                            initialValue:   IntegerConstantExpression(value: 0)
                        )
                        + channelnameVar = Variable(
                            type:   PrimitiveType.String
                            name:   unich.name + '_name'
                        )
        ports:          clsA.ports + clsB.ports - unich.sourcePort - unich.targetPort
        stateMachines:
            copy clsA.stateMachines into StateMachine*(
                vertices: smA.vertices + stateAReady = Vertex(name: 'AReady')
                transitions:
                    replace  each tr from sndTransitsA
                    in       this._old.transitions
                    by       [
                                copy tr into Transition(
                                    name:       'AReady'
                                    target:     stateAReady
                                    statements:
                                        replace  each snd from snds
                                        in       this._old.statements
                                        by       [
                                                    Assignment(
                                                        variable: channelnameVar
                                                        expression: StringConstantExpression(
                                                                    value: snd.signalName
                                                    ))
                                                    Assignment(
                                                        variable: ablVar,
                                                        expression: IntegerConstantExpression(value:1)
                                                    )
                                                ]
                                )
                                copy tr into Transition(
                                    name:       'Complete'
                                    source:     stateAReady
                                    statements: [
                                        BinaryOperatorExpression(
                                            operand1: VariableExpression(variable: ablVar)
                                            operator: Operator.equals
                                            operand2: IntegerConstantExpression(value: 2)
                                        )
                                        Assignment(  variable: ablVar
                                                     expression: IntegerConstantExpression(value: 3)
                                        )
                                        BinaryOperatorExpression(
                                            operand1: VariableExpression(variable: ablVar)
                                            operator: Operator.equals
                                            operand2: IntegerConstantExpression(value: 0)
                                        )
                                    ]
                                )
```

```
                                if tr.statements._count > tr.statements.collect(SendSignal snd)._count:
                                    copy tr into Transition(
                                        name:        'Cancel'
                                        source:      stateAReady
                                        statements: [
                                            Assignment(  variable: ablVar
                                                         expression: IntegerConstantExpression(value:0)
                                            )
                                        ]
                                    )
                            ]
                    )
                    + copy clsB.stateMachines into StateMachine*(
                        vertices: smB.vertices + stateBReady = Vertex(name: 'BReady')
                        transitions:
                            replace  each tr from sndTransitsB
                            in       this._old.transitions
                            by       [
                                        copy tr into Transition(
                                            name:        'BReady'
                                            target:      stateBReady
                                            statements:
                                                replace  each rcv from rcvs
                                                in       this._old.statements
                                                by       [
                                                            BinaryOperatorExpression(
                                                                operand1:
                                                                    BinaryOperatorExpression(
                                                                        operand1: VariableExpression(
                                                                                    variable: ablVar
                                                                                    )
                                                                        operator: Operator.equals
                                                                        operand2: IntegerConstantExpression(
                                                                                        value: 1
                                                                                    )
                                                                    )
                                                                operator: Operator.and
                                                                operand2:
                                                                    BinaryOperatorExpression(
                                                                        operand1: VariableExpression(
                                                                                    variable: channelnameVar
                                                                                    )
                                                                        operator: Operator.equals
                                                                        operand2: StringConstantExpression(
                                                                                        value: rcv.signalName
                                                                                    )
                                                                    )
                                                            )
                                                            Assignment(
                                                                variable: ablVar
                                                                expression: IntegerConstantExpression(value:2)
                                                            )
                                                        ]
                                        )
                                        copy tr into Transition(
                                            name:        'AcknowledgeCompletion'
                                            source:      stateBReady
                                            statements: [
                                                BinaryOperatorExpression(
                                                    operand1: VariableExpression(variable: ablVar)
                                                    operator: Operator.equals
                                                    operand2: IntegerConstantExpression(value: 3)
                                                )
                                                Assignment(  variable: ablVar
                                                             expression: IntegerConstantExpression(value: 0)
                                                )
                                            ]
                                        )
                                        copy tr into Transition(
                                            name:        'AcknowledgeCancel'
                                            source:      stateBReady
                                            target:      tr.source
                                            statements: [
                                                BinaryOperatorExpression(
                                                    operand1: VariableExpression(variable: ablVar)
                                                    operator: Operator.equals
                                                    operand2: IntegerConstantExpression(value: 0)
                                                )
                                            ]
                                        )
                                    ]
                    )
    )
```

Code fragment 39: *The* ***MergeObjects*** *transformation in* ***SLCOtrans v2****.*

Code fragment 40 shows the *MergeObjects* transformation in a traditional kind of model manipulation language, not based on pattern matching. The implementation is restricted to the assumption that classes can have only one state machine. This was done because it would take too much time to implement the transformation without this restriction, considering how long this took for SLCOtrans v2.

For this transformation, SLCOtrans v2 is slightly more concise than a traditional language. In the last two statements in **Code fragment 40**, it is indicated where in the abstract model objects and classes need to be added (or in this case: replaced). This does not need to be done in SLCOtrans v2. But compared to the total size of the implementations the difference this makes is negligible.

Taken into account that **Code fragment 40** still would have to be expanded to be equivalent to **Code fragment 39**, pattern matching does lead to more concise code in this case. It is clear though, that there are limits to what can currently be expressed in SLCOtrans v2 by the way the pattern matching concept in it is designed. Just like regular expressions, to which the design is conceptually similar, not everything can be expressed in it. The principle of pattern matching is convenient in the cases transformations can be expressed in them, but it lacks the flexibility of more granular control flow.

Code fragment 40 is a case too that could benefit from using concrete syntax. It could be improved by replacing the parts where `Assignment`s and `BinaryOperatorExpression`s (i.e. conditions) are used, by fragments in concrete syntax. The concrete syntax would be significantly shorter.

```
// 'each pair of state machines that are part of two communicating objects must communicate over a
// unique unidirectional, synchronous channel.'




// assume, for now: cls's have only 1 sm.

// arg $ch       // unidir sync ch. object connected to this $ch are merged.

// (model)...        // in EOL the model should be assigned here.

objA = $ch.sourceObject
objB = $ch.targetObject

clsA = objA.class
clsB = objB.class

smA = clsA.stateMachines.first        // only one
smB = clsB.stateMachines.first        // only one

clsAB = Class(
    name:            clsA.name    + '_'    + clsB.name
    variables:       clsA.variables       + clsB.variables
                            + ablVar = Variable(
                                type:            PrimitiveType.Integer
                                name:            $ch.name + '_abl'
                                initialValue:    IntegerConstantExpression(value: 0)
                            )
                            + channelnameVar = Variable(
                                type:    PrimitiveType.String
                                name:    $ch.name + '_name'
                            )
        ports:           clsA.ports + clsB.ports - $ch.sourcePort - $ch.targetPort
        stateMachines:
            smA
            smB
)

for transit in smA.transitions.clone:

    snds = transit.statements.collect(SendSignal snd)

    if snds > 0:

        stateAReady = Vertex(name: 'AReady')
        smA.vertices += stateAReady

        replace  transit
        in       smA.transitions
        by       [
                    copy transit into Transition(
                        name:        'AReady'
                        target:      stateAReady
                        statements: newStmts
                    )
                    copy transit into Transition(
                        name:        'Complete'
                        source:      stateAReady
                        statements: [
                            BinaryOperatorExpression(
                                operand1: VariableExpression(variable: ablVar)
                                operator: Operator.equals
                                operand2: IntegerConstantExpression(value: 2)
                            )
                            Assignment(variable: ablVar, expression: IntegerConstantExpression(value:3)
                            )
                            BinaryOperatorExpression(
                                operand1: VariableExpression(variable: ablVar)
                                operator: Operator.equals
                                operand2: IntegerConstantExpression(value: 0)
                            )
                        ]
                    )
                    if transit.statements.size = 1:
                        copy transit into Transition(
                            name:        'Cancel'
                            source:      stateAReady
                            statements: [
                                Assignment(  variable: ablVar
                                             expression: IntegerConstantExpression(value: 0)
                                )
                            ]
                        )
                ]
```

```
        for snd in snds:
            replace  snd
            in       newStmts
            by       [
                         Assignment(  variable: channelnameVar
                                      expression: StringConstantExpression(value: sigName)
                         )
                         Assignment(  variable: ablVar
                                      expression: IntegerConstantExpression(value: 1)
                         )
                     ]

for transit in smB.transitions.clone:

    rcvs = transit.statements.collect(SignalReception rcv)

    if rcvs > 0:

        stateBReady = Vertex(name: 'BReady')
        smB.vertices += stateBReady

        replace  transit
        in       smB.transitions
        by       [
                     copy transit into Transition(
                         name:        'BReady'
                         target:      stateBReady
                         statements: newStmts
                     )
                     copy transit into Transition(
                         name:        'AcknowledgeCompletion'
                         source:      stateBReady
                         statements: [
                             BinaryOperatorExpression(
                                 operand1: VariableExpression(variable: ablVar)
                                 operator: Operator.equals
                                 operand2: IntegerConstantExpression(value: 3)
                             )
                             Assignment(variable: ablVar, expression: IntegerConstantExpression(value:0)
                             )
                         ]
                     )
                     copy transit into Transition(
                         name:        'AcknowledgeCancel'
                         source:      stateBReady
                         target:      replTransit.source
                         statements: [
                             BinaryOperatorExpression(
                                 operand1: VariableExpression(variable: ablVar)
                                 operator: Operator.equals
                                 operand2: IntegerConstantExpression(value: 0)
                             )
                         ]
                     )
                 ]

        for rcv in rcvs:
            replace  rcv
            in       newStmts
            by       [
                             BinaryOperatorExpression(
                                 operand1:
                                     BinaryOperatorExpression(
                                         operand1: VariableExpression(variable: ablVar)
                                         operator: Operator.equals
                                         operand2: IntegerConstantExpression(value: 1)
                                     )
                                 operator: Operator.and
                                 operand2:
                                     BinaryOperatorExpression(
                                         operand1: VariableExpression(variable: channelnameVar)
                                         operator: Operator.equals
                                         operand2: StringConstantExpression(value: sigName)
                                     )
                             )
                             Assignment(variable: ablVar, expression: IntegerConstantExpression(value:2)
                             )
                     ]

replace  objA, objB
in       model.objects
by       Object(name: objA.name + '_' + objB.name, class: cls)

replace  clsA, clsB
in       model.classes
by       clsAB
```

Code fragment 40: *The* ***MergeObjects*** *transformation in* ***pseudocode****.*

8.13 Discussion

We now give a short overview of our findings. In **Table 1**, we show for each transformation we implemented whether it was more or less concise in SLCOtrans v2 than in the notation we used for traditional model manipulation languages, not based on pattern matching.

Section	Transformation	SLCOtrans v2 more (++) or less (--) concise
8.1	Add Delays	0
8.2	Strings to Integers	++
8.3	Identify Channels	0
8.4	Names to Arguments	0
8.5	Remove Unused Classes	+
8.6	Bidirectional to Unidirectional	+
8.7	Clone Classes	First: - Second: --
8.8	Merge Channels	+
8.9	Lossless to Lossy	? → ++ expected
8.10	Synchronous to Asynchronous	+
8.11	Exclusive Channels	--
8.12	Merge Objects	+

Table 1: *Comparison between the implementations in SLCOtrans v2 and non-pattern-matching languages. The meaning of the symbols is : **++** (much) more concise, **+** slightly more concise, **0** not much difference, **-** slightly less concise, **--** (much) less concise, **?** not implemented both.*

There are several elements that can make an implementation more concise in SLCOtrans v2. One element is that SLCOtrans v2 (and pattern matching languages in general) does not have to replace objects in every place they are referenced in the abstract model. This is used in the implementations for the transformations 'Strings to Integers', 'Bidirectional to Unidirectional', 'Merge Channels', 'Synchronous to Asynchronous', 'Exclusive Channels', 'Merge Objects'.

Another element is that SLCOtrans v2 has knowledge of the abstract model of SLCO, and automatically adds entities in the correct place in the abstract model, if possible. This is used in the implementations for the transformations 'Bidirectional to Unidirectional', 'Synchronous to Asynchronous', 'Exclusive Channels', 'Merge Objects'.

SLCOtrans v2 can also collect all occurrences of values of enums and primitive types (e.g. integers, strings, booleans) as opposed to typical model manipulation languages, such as EOL. This is used in the implementations for the transformations 'Strings to Integers',

Finally, SLCOtrans v2 has the option to use fragments of SLCO in it. This is used in the transformation 'Lossless to Lossy' and could be used in 'Merge Objects'.

Transformations such as the 'Exclusive Channels' show though, that as transformations become more complex, they become increasingly harder to write (and understand) in SLCOtrans v2. They can also become longer fast if the transformation is too complex. It is even likely, since the concept of pattern matching is similar to regular expressions, that some transformations are impossible to express in an extended version of SLCOtrans v2.

The principle of pattern matching is convenient in the cases transformations can be expressed in them, but it lacks the flexibility of more granular control flow.

9 Conclusions

In this paper, we have investigated how convenient DSTLs are to use in practice.

We have found that DSTLs are practical for relatively simple transformations, but that as the transformations become more complex, the advantages quickly diminish and it becomes easier to use more traditional paradigms based on the abstract syntax (i.e. traditional general purpose transformation languages) than DSTLs.

The main problem with DSTLs is that transformations are often complex in practice. Using the concrete syntax might make transformations slightly easier to understand for domain experts, but this will often not be enough, because in many cases the transformations in practice themselves are too complex to understand and write by the domain experts, who typically have little programming experience.

DSTLs work well when there are large fragments of the source- or target DSLs that can be used in the DSTL virtually unchanged, but it delivers no advantage, or even extra inconvenience, for more complex transformations where this is not the case.

We also compared a language based on pattern matching with (a representation of) traditional model manipulation languages, not based on the concept of pattern matching. The main advantage of this is that elements do not have to be replaced separately in all places they are referenced. This does only provide a slight difference though. The concept of pattern matching is intuitive to humans, but it is not as flexible as the control flow in traditional languages. Therefore, there are limitations to what can be expressed with them, and they become very complex quite quickly.

10 Future work

Though we have shown that the use of DSTLs is limited in many cases, there are also cases where they *are* more convenient than GPTLs. To be able to use the language introduced in this paper (SLCOtrans v2) for these cases, a number of steps need to be taken.

There has not yet been created a machine processable (formal) grammar for SLCOtrans v2. This can be done using Xtext [26, 27].

Also, the (formal) semantics of SLCOtrans v2 have not yet been implemented. This can for example be done using a suitable GPTL or EOL [18, 19, 21]. It can also be done using a DSL which describes how to derive a DSTL from its corresponding DSL.

Of course, the method used in this paper to create a DSTL is not limited to SLCOtrans. DSTLs can be created for other DSLs too. To create a DSTL for a different DSL, an (Xtext) grammar needs to be created specifically for that DSL, and the semantics need to be specifically implemented for that DSL too (in EOL, for example).

Using the technique used in this paper, constructs from the language in which the semantics of the DSTL are implemented, are not automatically inherited by the DSTL. This means every construct of the implementing language needs to be mapped separately onto the DSTL for it to be used in the DSTL. Specifically interesting would be to investigate a way to automatically enable using constructs and features from the implementing language in the DSTL.

The benefits of DSLs have confirmed by been empirical research [41]. However, additionally to DSL notation, DSTLs also include syntax specific to transformations. Therefore, additional empirical

research is needed to show whether domain still provide advantages to domain experts despite the syntax specific to transformations.

10.1 Transformation verification

Model transformations can contain errors, just like any other artifact created by humans. Therefore it is necessary to be able to verify them. Sander de Putter and Anton Wijs have "[verified] an existing approach for checking property-preservation for model transformations that may affect synchronising behaviour of parallel processes" [42, 43]. Such a verification technique that checks property-preservation for model transformations could be used to verify SLCOtrans.

In order to use this method of checking property-preservation for model transformations, the transformation needs to be mapped onto a transformation on state spaces. The less complex the syntax of a language is, the easier it is to map onto transition systems. Therefore mapping a DSTL could be a first step for mapping and verifying a language with a more complex syntax.

11 Acknowledgements

I would like to thank my supervisors: Anton Wijs and Sander de Putter, for supporting me during this project, for their advice, support, and discussions.

I would also like to thank Sander, as well as Yaping (Luna) Luo, for helping me when I had trouble with Xtext, Epsilon, and Eclipse Modeling Tools.

Finally I would like to thank everyone not mentioned here that supported me in any way while working or not working on this project.

12 Credits

Figure 1 was partially inspired by a figure provided by Sander de Putter.

Code fragment 1 and **Code fragment 3** were created by Sander de Putter.

Figure 2, **Figure 3**, **Figure 7**, **Figure 9**, and **Code fragment 2** (without syntax highlighting) were created by Luc Engelen.

The initial design (mockup) of SLCOtrans was created by Sander de Putter.

13 References

[1] D. C. Schmidt, "Guest Editor's Introduction: Model-Driven Engineering," *IEEE Computer,* vol. 39, no. 2, pp. 25-31, 2006.

[2] J. Irazábal, C. Pons and C. Neil, "Model transformation as a mechanism for the implementation of domain specific transformation languages," *EJS: SADIO Electronic Journal of Informatics and Operations Research,* vol. 9, no. 1, pp. 49-66, 2010.

[3] L. Silvestre, "A Domain Specific Transformation Language to Support the Interactive Definition of Model Transformation Rules," 2014.

[4] L. Hermerschmidt, K. Hölldobler, B. Rumpe and A. Wortmann, "Generating Domain-Specific Transformation Languages for Component & Connector Architecture Descriptions," *CEUR Workshop Proceedings,* vol. 1463, pp. 18-23, 2015.

[5] J. Rumbaugh, I. Jacobson and G. Booch, Unified Modeling Language Reference Manual, The, Pearson Higher Education, 2004.

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

[7] G. J. Holzmann, "The Model Checker SPIN," *IEEE Transactions on Software Engineering - Special issue on formal methods in software practice,* vol. 23, no. 5, pp. 279-295, 1997.

[8] A. v. Deursen and P. Klint, "Domain–Specific Language Design Requires Feature Descriptions," *Journal of Computing and Information Technology,* vol. 10, pp. 1-17, 2002.

[9] A. V. Deursen, P. Klint and J. Visser, "Domain-Specific Languages: An Annotated Bibliography.," *Sigplan Notices,* vol. 35, no. 6, pp. 26-36, 2000.

[10] M. Bernardo, V. Cortellessa and A. (. Pierantonio, Formal Methods for Model-Driven Engineering. 12th International School on Formal Methods for the Design of Computer, Communication and Software Systems, SFM 2012, Bertinoro, Italy, June 18-23, 2012. Advanced Lectures, Springer-Verlag Berlin Heidelberg, 2012.

[11] I. Damyanov and M. Sukalinska, "Domain Specific Languages in Practice," *International Journal of Computer Applications (0975-8887),* vol. 115, no. 2, pp. 42-45, 2015.

[12] M. v. Amstel, S. Andova, M. v. d. Brand and L. Engelen, "Simple Language of Communicating Objects," 27 May 2013. [Online]. Available: http://www.win.tue.nl/~lengelen/slco/SLCO.pdf.

[13] L. Engelen, From Napkin Sketches to Reliable Software, Technische Universiteit Eindhoven, 2012.

[14] J. S. Cuadrado, E. Guerra and J. d. Lara, "Towards the Systematic Construction of Domain-Specific Transformation Languages," *LNCS 8569, ECMFA 2014,* pp. 196-212, 2014.

[15] B. Rumpe and I. Weisemöller, "A Domain Specific Transformation Language," *CoRR / ME: Models and Evolution,* 2011.

[16] D. Baum, Dave Baum's Definitive Guide to Lego Mindstorms, Apress, 2000.

[17] "ATL Transformation Language," [Online]. Available: https://eclipse.org/atl/.

[18] "Epsilon Object Language," [Online]. Available: http://www.eclipse.org/epsilon/doc/eol/.

[19] D. S. Kolovos, R. F. Paige and F. A. Polack, "The Epsilon Object Language (EOL)," *LNCS 4066, ECMDA-FA 2006,* pp. 128-142, 2006.

[20] "Epsilon," [Online]. Available: http://www.eclipse.org/epsilon/.

[21] D. Kolovos, L. Rose, A. García-Domínguez and R. Paige, "The Epsilon Book," 2015. [Online]. Available: http://www.eclipse.org/epsilon/doc/book/.

[22] R. F. Paige, D. S. Kolovos, L. M. Rose, N. Drivalos and F. A. C. Polack, "The Design of a Conceptual Framework and Technical Infrastructure for Model Management Language Engineering," *ICECCS '09 Proceedings of the 2009 14th IEEE International Conference on Engineering of Complex Computer Systems,* pp. 162-171, 2009.

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

[24] D. Baum, "NQC Programmer's Guide," 2003. [Online]. Available: http://bricxcc.sourceforge.net/nqc/.

[25] A. Pnueli, "The Temporal Logic of Programs," *Proceedings fo the 18th Annual Symposium on Foundations of Computer Science,* 1977.

[26] "Xtext," [Online]. Available: https://eclipse.org/Xtext/.

[27] M. Eysholdt and H. Behrens, "Xtext: implement your language faster than the quick and dirty way.," *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion,* 2010.

[28] B. Courcelle, "Graph rewriting: An algebraic and logic approach.," in *Handbook of Theoretical Computer Science, volume B*, 1990, pp. 193-242.

[29] H. Ehrig, R. Heckel, M. Korff, M. Lowe, L. Ribeiro, A. Wagner and A. Corradini, "Algebraic Approaches To Graph Transformation Part II: Single Pushout Approach And Comparison With Double Pushout Approach," in *Handbook of graph grammars and computing by graph transformation*, NJ, USA, World Scientific Publishing Co., Inc. River Edge, 1997, pp. 247-312.

[30] T. Levendovszky, L. Lengyel and H. Charaf, "Extending the dpo approach for topological validation of metamodel-level graph rewriting rules.," *WSEAS Transactions on Information Science and Applications,* vol. 2, no. 2, pp. 226-231, 2005.

[31] R. Heckel and A. Cherchago, "Application of Graph Transformation for Automating Web Service Discovery.," *Dagstuhl Seminar Proceedings. Schloss Dagstuhl-Leibniz-Zentrum fr Informatik.,* 2005.

[32] K. Czarnecki and S. Helsen, "Classification of model transformation approaches," *Proceedings of the 2nd OOPSLA Workshop on Generative Techniques in the Context of the Model Driven Architecture,* vol. 45, no. 3, 2003.

[33] "Epsilon Transformation Language," [Online]. Available: http://www.eclipse.org/epsilon/doc/etl/.

[34] D. S. Kolovos, R. F. Paige and F. A. C. Polack, "The Epsilon Transformation Language," *ICMT 2008, LNCS 5063,* pp. 46-60, 2008.

[35] "Eclipse Modeling Tools," [Online]. Available: http://www.eclipse.org/downloads/packages/eclipse-modeling-tools/neonm6.

[36] "Eclipse Modeling Framework," [Online]. Available: https://eclipse.org/modeling/emf/.

[37] D. Steinberg, F. Budinsky, E. Merks and M. Paternostro, EMF: Eclipse Modeling Framework, 2009.

[38] "Apache Ant," [Online]. Available: http://ant.apache.org/.

[39] C. J. Date and H. Darwen, A Guide To Sql Standard, Addison-Wesley, 1997.

[40] "Xtend," [Online]. Available: http://www.eclipse.org/xtend/.

[41] K. Tomaž, O. Nuno, M. Marjan, P. V. J. Maria, Č. Matej, D. C. Daniela and H. R. Pedro, "Comparing general-purpose and domain-specific languages: An empirical study," *Computer Science and Information Systems,* vol. 7, no. 2, pp. 247-264, 2010.

[42] S. d. Putter, On the formal correctness of a model transformation verification technique, Technische Universiteit Eindhoven, 2014.

[43] S. d. Putter and A. Wijs, "Verifying a Verifier: On the Formal Correctness of an LTS Transformation Verification Technique," *Proc. 19th International Conference on Fundamental Approaches to Software Engineering (FASE'16), Eindhoven, The Netherlands, volume 9633 of Lecture Notes in Computer Science, pp. 383-400, Springer (2016).*

14 Appendices

14.1 Appendix A: The implementation of SLCOtrans

```
// inplace

'EXECUTING transform.eol...\n\n'.println;

var Slco  = TextualSlco;      // In
var Tr    = TextualSlcoTrans;
//var Out = TextualSlcoOut;

var model_       = Slco!Model.all.first;            // eq to: var `model` : Model = ... ! ... .all.selectOne(it|true);
var modelTrans   = Tr!ModelTransformation.all.first;

// var Slco_BiChs = Slco!BidirectionalChannel.all;
// for some reason 'var Slco_BiCh  = Slco!BidirectionalChannel' wont work in ': Slco_BiCh' but will work in 'Slco_BiCh.all':
var Slco_BiCh = Slco!BidirectionalChannel;  // i find it clearer to keep the .all explicit.
var Tr_BiCh   = Tr!BidirectionalChannel;
var Tr_UniCh  = Tr!UnidirectionalChannel;


var portsKeep = OrderedSet{};    // Ports to keep after transfo (because they are still used).


// select first~ Tr_BiChL.
var chL = modelTrans.channelsL.selectOne(it|it.isKindOf(Tr!BidirectionalChannel));
("chL: " + chL.name).println;

// feach chIn:
for (chIn : Slco!BidirectionalChannel in model_.channels.clone) {   // Slco_BiCh is/o Slco!BidirectionalChannel didnt work here, i think.
                                                                    // for (chIn in Slco_BiCh.all) {  // .clone b/c rm (/add) (ch) in loop.
                                                                    // assume only bich for now. (b/c of attrs.)
  ("-chIn: " + chIn.name).println;

  // bich is always a match. (when not bich: check if match.)

  // add new chs:
  for (chR : Tr!UnidirectionalChannel in modelTrans.channelsR) {    // assume only unich for now. (b/c of attrs.)
    (" -chR: " + chR.name).println;

    // cr new, b/c (Tr!Uni != Slco!Uni) unfortunately:          // (Tr!Ch subclasses have no obj's, so was neccesary anyway.)
    var chOut : new Slco!UnidirectionalChannel;
    chOut.name = chIn.name + loopCount.asString;            // dont use chR nm for out, b/c chL nm is not used to match either.

    // (simply) cp attrs:
      chOut.argumentTypes  = chR.argumentTypes;

      //chOut.channelType  = chR.channelType;:
      switch (chR.channelType.asString) {              // no fall through.
        case "async_lossless":
            chOut.channelType = TextualSlco!ChannelType#AsynchronousLossless;  // 'Slco!' doesnt work here for some reason.
        case "async_lossy":
```

```
50.             chOut.channelType = TextualSlco!ChannelType#AsynchronousLossy;    // 'Slco!' doesnt work here for some reason.
51.           case "sync":
52.             chOut.channelType = TextualSlco!ChannelType#Synchronous;          // 'Slco!' doesnt work here for some reason.
53.           default:
54.             ("ERROR: unknown Tr.ChannelType: " + chR.channelType).println;
55.         }
56.
57.
58.       // chOut.{src tgt}x(obj) = (chIn/chR?).(obj)x(1 2):
59.
60.
61.       // ASSUMPTIONS/REQUIREMENTS/PRECONDITIONS:
62.       // - Each portR follows from 1 portL:      not < 1: dont know (any) obj/cls. Not > 1: dont know (which one) obj/cls.
63.       // - Therefore (for now): all portsL's must be different.
64.       // - ...
65.
66.       // - Cr new prt if needed.
67.       // - Nm prt.
68.       // - Place prt in correct obj/cls + ch.
69.
70.       // Source-(port/obj):
71.       ("  -sourcePort: " + chR.sourcePort.name).println;
72.       if (chL.port1.name.isSubstringOf(chR.sourcePort.name)) {      // If chL.port1nm in chR.srcPortNm:
73.
74.         chOut.sourceObject = chIn.object1;                  // srcObj = obj1.
75.
76.         if (chL.port1.name = chR.sourcePort.name) { // If chL.port1nm = chR.srcPortNm: (IE: if nm equal.):
77.                                                     // case not strictly neccesary-> 'else' result in same thing, but 'faster' + clearer what actually happens.
78.
79.           chOut.sourcePort = chIn.port1;                    // Use old port: srcPrt = prt1.
80.           // ->                                  // This prt is the same one as all prts in this obj/cls w/ same nm.
81.
82.         } else {                                // If chL.port1nm != chR.srcPortNm:
83.
84.           var newName =                             // New nm = repl chL.nm by chIn.nm in chR.nm. IE: paste chIn.port1nm in chR.srcPortNm.
85.             chR.sourcePort.name
86.             .replace(chL.port1.name, chIn.port1.name);
87.
88.           // mk a new port ONLY if the new one doesnt exist yet IN SAME CLS!:
89.
90.           var portsSameNameSameClass =                    // ports w/ same nm as new nm + same cls.
91.             chOut.sourceObject.class.ports.select(it|it.name = newName);  // (chOut.sourceObject same as chIn.object1).
92.           // ->
93.           if (portsSameNameSameClass.isEmpty) {               // If no prt w/ same nm + cls yet:
94.             chOut.sourcePort = new Slco!Port;                 // Cr new chOut.srcPort.
95.             chOut.sourcePort.name =  newName;                 // assign th new nm.
96.             chOut.sourceObject.class.ports.add(chOut.sourcePort);        // add th port in obj.cls.ports.
97.
98.           } else if (portsSameNameSameClass.size = 1) {           // If 1 prt w/ same nm + cls already exists: (This happens e/g when this nm occurs more
99.                                                               // often in chR. But also when both resolved nm and cls are same):
100.            chOut.sourcePort = portsSameNameSameClass.first;          // Use it. (first is th only one.)
101.
102.          } else {                                // If more prts w/ same nm:
103.            ("ERROR: port" +portsSameNameSameClass.first.name+
104.            " exists multiple times in class " +chIn.object1.class+
105.            ". Not supposed to happen!").println;               // Notify: 'should not happen'.
106.          }
```

```
107.        }
108.
109.        portsKeep.add(chOut.sourcePort);                    // keep port.
110.
111.      } else if (chL.port2.name.isSubstringOf(chR.sourcePort.name)) {      // If chL.port2nm in chR.srcPortNm:
112.
113.        chOut.sourceObject = chIn.object2;                    // srcObj = obj2.
114.
115.        if (chL.port2.name = chR.sourcePort.name) {              // If chL.port2nm = chR.srcPortNm: (IE: if nm equal.):  // case not strictly neccesary->
116.                                                                 // 'else' result in same thing, but 'faster' + clearer what actually happens.
117.
118.          chOut.sourcePort = chIn.port2;                    // Use old port: srcPrt = prt2.
119.          // ->                                  // This prt is the same one as all prts in this obj/cls w/ same nm.
120.
121.        } else {                                // If chL.port2nm != chR.srcPortNm:
122.
123.          var newName =                                // New nm = repl chL.nm by chIn.nm in chR.nm. IE: paste chIn.port2nm in chR.srcPortNm.
124.            chR.sourcePort.name
125.            .replace(chL.port2.name, chIn.port2.name);
126.
127.          // mk a new port ONLY if the new one doesnt exist yet IN SAME CLS!:
128.
129.          var portsSameNameSameClass =                    // ports w/ same nm as new nm + same cls.
130.            chOut.sourceObject.class.ports.select(it|it.name = newName);  // (chOut.sourceObject same as chIn.object2).
131.          // ->
132.          if (portsSameNameSameClass.isEmpty) {                // If no prt w/ same nm + cls yet:
133.            chOut.sourcePort = new Slco!Port;                  // Cr new chOut.srcPort.
134.            chOut.sourcePort.name =  newName;                  // assign th new nm.
135.            chOut.sourceObject.class.ports.add(chOut.sourcePort);        // add th port in obj.cls.ports.
136.
137.          } else if (portsSameNameSameClass.size = 1) {            // If 1 prt w/ same nm + cls already exists: (This happens e/g when this nm occurs more
138.                                                                   // often in chR. But also when both resolved nm and cls are same):
139.            chOut.sourcePort = portsSameNameSameClass.first;          // Use it. (first is th only one.)
140.
141.          } else {                                // If more prts w/ same nm:
142.            ("ERROR: port" +portsSameNameSameClass.first.name+
143.            " exists multiple times in class " +chIn.object2.class+
144.            ". Not supposed to happen!").println;                // Notify: 'should not happen'.
145.          }
146.        }
147.
148.        portsKeep.add(chOut.sourcePort);                    // keep port.
149.
150.      } else {
151.        "chR.sourcePort.nm does not (and should) contain chL.port1.nm or chL.port2.nm".println();
152.      }
153.
154.      // Target-(port/obj):
155.      ("  -targetPort: " + chR.targetPort.name).println;
156.      if (chL.port1.name.isSubstringOf(chR.targetPort.name)) {      // If chL.port1nm in chR.tgtPortNm:
157.
158.        chOut.targetObject = chIn.object1;                    // tgtObj = obj1.
159.
160.        if (chL.port1.name = chR.targetPort.name) {              // If chL.port1nm = chR.tgtPortNm: (IE: if nm equal.):  // case not strictly neccesary->
161.                                                                 // 'else' result in same thing, but 'faster' + clearer what actually happens.
162.
163.          chOut.targetPort = chIn.port1;                    // Use old port: tgtPrt = prt1.
```

```
164.          // ->                                  // This prt is the same one as all prts in this obj/cls w/ same nm.
165.
166.        } else {                                 // If chL.port1nm != chR.tgtPortNm:
167.
168.          var newName =                             // New nm = repl chL.nm by chIn.nm in chR.nm. IE: paste chIn.port1nm in chR.tgtPortNm.
169.            chR.targetPort.name
170.            .replace(chL.port1.name, chIn.port1.name);
171.
172.          // mk a new port ONLY if the new one doesnt exist yet IN SAME CLS!:
173.
174.          var portsSameNameSameClass =                    // ports w/ same nm as new nm + same cls.
175.            chOut.targetObject.class.ports.select(it|it.name = newName);  // (chOut.targetObject same as chIn.object1).
176.          // ->
177.          if (portsSameNameSameClass.isEmpty) {                // If no prt w/ same nm + cls yet:
178.            chOut.targetPort = new Slco!Port;                  // Cr new chOut.tgtPort.
179.            chOut.targetPort.name =  newName;                  // assign th new nm.
180.            chOut.targetObject.class.ports.add(chOut.targetPort);        // add th port in obj.cls.ports.
181.
182.          } else if (portsSameNameSameClass.size = 1) {            // If 1 prt w/ same nm + cls already exists: (This happens e/g when this nm occurs more
183.                                                                   // often in chR. But also when both resolved nm and cls are same):
184.            chOut.targetPort = portsSameNameSameClass.first;          // Use it. (first is th only one.)
185.
186.          } else {                                 // If more prts w/ same nm:
187.            ("ERROR: port" +portsSameNameSameClass.first.name+
188.            " exists multiple times in class " +chIn.object1.class+
189.            ". Not supposed to happen!").println;                // Notify: 'should not happen'.
190.          }
191.        }
192.
193.        portsKeep.add(chOut.targetPort);                   // keep port.
194.
195.      } else if (chL.port2.name.isSubstringOf(chR.targetPort.name)) {      // If chL.port2nm in chR.tgtPortNm:
196.
197.        chOut.targetObject = chIn.object2;                   // tgtObj = obj2.
198.
199.        if (chL.port2.name = chR.targetPort.name) {              // If chL.port2nm = chR.tgtPortNm: (IE: if nm equal.):  // case not strictly neccesary->
200.                                                                 // 'else' result in same thing, but 'faster' + clearer what actually happens.
201.
202.          chOut.targetPort = chIn.port2;                   // Use old port: tgtPrt = prt2.
203.          // ->                                  // This prt is the same one as all prts in this obj/cls w/ same nm.
204.
205.        } else {                                 // If chL.port2nm != chR.tgtPortNm:
206.
207.          var newName =                             // New nm = repl chL.nm by chIn.nm in chR.nm. IE: paste chIn.port2nm in chR.tgtPortNm.
208.            chR.targetPort.name
209.            .replace(chL.port2.name, chIn.port2.name);
210.
211.          // mk a new port ONLY if the new one doesnt exist yet IN SAME CLS!:
212.
213.          var portsSameNameSameClass =                    // ports w/ same nm as new nm + same cls.
214.            chOut.targetObject.class.ports.select(it|it.name = newName);  // (chOut.targetObject same as chIn.object2).
215.          // ->
216.          if (portsSameNameSameClass.isEmpty) {                // If no prt w/ same nm + cls yet:
217.            chOut.targetPort = new Slco!Port;                  // Cr new chOut.tgtPort.
218.            chOut.targetPort.name =  newName;                  // assign th new nm.
219.            chOut.targetObject.class.ports.add(chOut.targetPort);        // add th port in obj.cls.ports.
220.
```

```
221.          } else if (portsSameNameSameClass.size = 1) {            // If 1 prt w/ same nm + cls already exists: (This happens e/g when this nm occurs more
222.                                                                   // often in chR. But also when both resolved nm and cls are same):
223.            chOut.targetPort = portsSameNameSameClass.first;          // Use it. (first is th only one.)
224.
225.          } else {                                 // If more prts w/ same nm:
226.            ("ERROR: port" +portsSameNameSameClass.first.name+
227.            " exists multiple times in class " +chIn.object2.class+
228.            ". Not supposed to happen!").println;               // Notify: 'should not happen'.
229.          }
230.        }
231.
232.        portsKeep.add(chOut.targetPort);                    // keep port.
233.
234.      } else {
235.        "chR.targetPort.nm does not (and should) contain chL.port1.nm or chL.port2.nm".println();
236.      }
237.
238.      // add new chs:
239.      model_.channels.add(chOut);                        // add new chs.
240.
241.    }                                    // (feach chR.)
242.
243.    // rm old ch:
244.    model_.channels.remove(chIn);                      // cannot just simply adapt chIn, b/c there might be multiple chOut.
245.
246.  }                                    // (feach chIn.)
247.
248.
249.  ////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
250.
251.
252.  // feach transfo (in Tr):
253.  for (smTransfo : Tr!StateMachineTransformation in modelTrans.transformations) {             // for (smTransfo in Tr!StateMachineTransformation.all) {
254.    ("\n"+"smTransfo (in Tr):").println;
255.
256.    // feach classIn:
257.    for (clsIn : Slco!Class in model_.classes) {                              // for (clsIn in Slco!Class.all) {
258.      ("-clsIn: " + clsIn.name).println;
259.
260.      // feach smIn:
261.      for (smIn : Slco!StateMachine in clsIn.stateMachines) {
262.        (" -smIn: " + smIn.name).println;
263.
264.
265.        if (
266.          smTransfo.stateMachineL.transitions.first.source.name = smIn.transitions.first.source.name and
267.          smTransfo.stateMachineL.transitions.first.target.name = smIn.transitions.first.target.name and
268.          smTransfo.stateMachineL.transitions.second.source.name = smIn.transitions.second.source.name and
269.          smTransfo.stateMachineL.transitions.second.target.name = smIn.transitions.second.target.name and
270.          (
271.            (
272.              smTransfo.stateMachineL.transitions.first.statements.first.isKindOf(Tr!SendSignal) and
273.              smIn.transitions.first.statements.first.isKindOf(Slco!SendSignal) and
274.              smTransfo.stateMachineR.transitions.first.statements.first.isKindOf(Tr!SendSignal) and      // smL matches smIn.
275.              smTransfo.stateMachineL.transitions.second.statements.first.isKindOf(Tr!SignalReception) and
276.              smIn.transitions.second.statements.first.isKindOf(Slco!SignalReception) and
277.              smTransfo.stateMachineR.transitions.second.statements.first.isKindOf(Tr!SignalReception)    // smL matches smIn.
```

```
278.            ) or (
279.              smTransfo.stateMachineL.transitions.first.statements.first.isKindOf(Tr!SignalReception) and
280.              smIn.transitions.first.statements.first.isKindOf(Slco!SignalReception) and
281.              smTransfo.stateMachineR.transitions.first.statements.first.isKindOf(Tr!SignalReception) and  // smL matches smIn.
282.              smTransfo.stateMachineL.transitions.second.statements.first.isKindOf(Tr!SendSignal) and
283.              smIn.transitions.second.statements.first.isKindOf(Slco!SendSignal) and
284.              smTransfo.stateMachineR.transitions.second.statements.first.isKindOf(Tr!SendSignal)        // smL matches smIn.
285.            )
286.          )
287.
288.        ){
289.
290.          ("  -smL matches smIn. + smR same as smL.").println;
291.
292.          // 1st transit:
293.          var newNameFor1stTransit =
294.            smTransfo.stateMachineR.transitions.first.statements.first.port.name
295.            .replace(
296.              smTransfo.stateMachineL.transitions.first.statements.first.port.name,
297.              smIn.transitions.first.statements.first.port.name
298.            );
299.
300.          // ASSUME needed ports already exist.
301.
302.          var portsSameNameSameClass1 =
303.            clsIn.ports.select(it|it.name = newNameFor1stTransit);
304.          // ->
305.          if (portsSameNameSameClass1.isEmpty) {                // If no prt w/ same nm + clsIn yet:
306.            smIn.transitions.first.statements.first.port = new Slco!Port;    // Cr new
307.            smIn.transitions.first.statements.first.port.name =  newNameFor1stTransit;  // assign th new nm.
308.            clsIn.ports.add(smIn.transitions.first.statements.first.port);    // add th port.
309.
310.          } else if (portsSameNameSameClass1.size = 1) {            // If 1 prt w/ same nm + clsIn already exists:
311.            smIn.transitions.first.statements.first.port = portsSameNameSameClass1.first;  // Use it. (first is th only one.)
312.
313.          } else {                                  // If more prts w/ same nm:
314.            ("ERROR: port" +portsSameNameSameClass1.first.name+
315.            " exists multiple times in class " +clsIn+
316.            ". Not supposed to happen!").println;               // Notify: 'should not happen'.
317.          }
318.
319.          portsKeep.add(smIn.transitions.first.statements.first.port);
320.
321.
322.          // 2nd transit:
323.          var newNameFor2ndTransit =
324.            smTransfo.stateMachineR.transitions.second.statements.first.port.name
325.            .replace(
326.              smTransfo.stateMachineL.transitions.second.statements.first.port.name,
327.              smIn.transitions.second.statements.first.port.name
328.            );
329.
330.          // ASSUME needed ports already exist.
331.
332.          var portsSameNameSameClass2 =
333.            clsIn.ports.select(it|it.name = newNameFor2ndTransit);
334.          // ->
```

```
335.          if (portsSameNameSameClass2.isEmpty) {                // If no prt w/ same nm + clsIn yet:
336.            smIn.transitions.second.statements.first.port = new Slco!Port;    // Cr new
337.            smIn.transitions.second.statements.first.port.name =  newNameFor2ndTransit;    // assign th new nm.
338.            clsIn.ports.add(smIn.transitions.second.statements.first.port);    // add th port.
339.
340.          } else if (portsSameNameSameClass2.size = 1) {            // If 1 prt w/ same nm + clsIn already exists:
341.            smIn.transitions.second.statements.first.port = portsSameNameSameClass2.first;  // Use it. (first is th only one.)
342.
343.          } else {                                // If more prts w/ same nm:
344.            ("ERROR: port" +portsSameNameSameClass2.first.name+
345.            " exists multiple times in class " +clsIn+
346.            ". Not supposed to happen!").println;               // Notify: 'should not happen'.
347.          }
348.
349.          portsKeep.add(smIn.transitions.second.statements.first.port);
350.
351.        }
352.      }
353.    }
354.  }
355.
356.
357.
358.  ////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
359.
360.  // rm unused prts.
361.  for (cls in Slco!Class.all) {
362.    cls.ports = cls.ports.select(it|portsKeep.includes(it));
363.  }
364.
```

Code fragment 41: The implementation of SLCOtrans. *These are the semantics of SLCOtrans, implemented in the Epsilon Object Language (EOL) [18, 19, 21]*