Final Report on Combining SysML and CML

Document Number: D22.4
Version: 1.0
Date: March 2013
Public Document

http://www.compass-research.eu
Contributors:

Alvaro Miyazawa, York
Ana Cavalcanti, York
Juliano Iyoda, UFPE
Márcio Cornélio, UFPE
Lucas Albertins, UFPE
Richard Payne, UNEW

Editors:

Alvaro Miyazawa, York

Reviewers:

Ralph Hains, Atego
Stefan Hallerstede, Aarhus University
Klaus Kristensen, B&O
### Document History

<table>
<thead>
<tr>
<th>Ver</th>
<th>Date</th>
<th>Author</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00</td>
<td>30-07-2012</td>
<td>Alvaro Miyazawa</td>
<td>Initial document version.</td>
</tr>
<tr>
<td>0.01</td>
<td>20-11-2012</td>
<td>Alvaro Miyazawa</td>
<td>Included material on guidelines and meta language.</td>
</tr>
<tr>
<td>0.02</td>
<td>06-12-2012</td>
<td>Alvaro Miyazawa</td>
<td>Updated with Ana’s comments.</td>
</tr>
<tr>
<td>0.03</td>
<td>10-12-2012</td>
<td>Alvaro Miyazawa</td>
<td>Included the semantics of simple blocks.</td>
</tr>
<tr>
<td>0.04</td>
<td>12-12-2012</td>
<td>Alvaro Miyazawa</td>
<td>Included the semantics of state machine diagrams.</td>
</tr>
<tr>
<td>0.05</td>
<td>16-01-2013</td>
<td>Lucas Albertins</td>
<td>Included the semantics of sequence diagrams.</td>
</tr>
<tr>
<td>0.06</td>
<td>21-12-2012</td>
<td>Alvaro Miyazawa</td>
<td>Included the semantics of ports, and added SysML example in Chapter 2.</td>
</tr>
<tr>
<td>0.07</td>
<td>28-12-2012</td>
<td>Alvaro Miyazawa</td>
<td>Included the semantics of composite blocks, and completed SysML example in Chapter 2.</td>
</tr>
<tr>
<td>0.08</td>
<td>30-12-2012</td>
<td>Alvaro Miyazawa</td>
<td>Reviewed guidelines and example section.</td>
</tr>
<tr>
<td>0.09</td>
<td>31-12-2012</td>
<td>Alvaro Miyazawa</td>
<td>Reviewed and completed conclusions.</td>
</tr>
<tr>
<td>0.10</td>
<td>31-12-2012</td>
<td>Alvaro Miyazawa</td>
<td>Completed structural diagrams chapter.</td>
</tr>
<tr>
<td>0.11</td>
<td>05-02-2013</td>
<td>Márcio Cornéllo</td>
<td>Included the semantics of activity diagrams.</td>
</tr>
<tr>
<td>0.12</td>
<td>22-02-2013</td>
<td>Alvaro Miyazawa</td>
<td>First review of parts of introduction and structural diagrams.</td>
</tr>
<tr>
<td>0.13</td>
<td>28-02-2013</td>
<td>Alvaro Miyazawa</td>
<td>Review of state machine diagrams.</td>
</tr>
<tr>
<td>0.14</td>
<td>07-03-2013</td>
<td>Alvaro Miyazawa</td>
<td>Review of parts of introduction, structural and state machine diagrams, and conclusions.</td>
</tr>
<tr>
<td>0.15</td>
<td>27-03-2013</td>
<td>Alvaro Miyazawa</td>
<td>Addressed comments from the internal reviewers.</td>
</tr>
<tr>
<td>1.0</td>
<td>28-03-2013</td>
<td>Alvaro Miyazawa</td>
<td>Final version ready for submission.</td>
</tr>
</tbody>
</table>
Abstract

The Systems Modeling Language (SysML) is becoming a de-facto standard in the systems engineering community, and has been considered as a potential graphical notation for the design and analysis of Systems of Systems. This deliverable aims at providing a formal account to a representative subset of SysML to support formal analysis of models specified in this notation.

This goal is achieved by means of a formal semantics of SysML diagrams, which in this work is defined in terms of Compass Modelling Language (CML). The choice of CML as the semantic domain for SysML models is motivated by the ability to explore the CML theories and tools in the analysis of the models. In this report we target three SysML behavioural diagrams, namely, activity, sequence and state machine diagrams, and two structural diagrams, namely, block definition diagrams and internal block diagrams. The semantics of blocks in fact gives an integrated semantics of multi-diagram SysML models.

We identify a set of guidelines of usage for SysML diagrams that support the definition of a self-contained semantics of SysML models. Additionally, we identify a subset of diagram constructs that are more frequently used. Based on these guidelines, we propose a formal denotational semantics of SysML models. The semantics is defined by an inductive function from the abstract syntax of SysML to the abstract syntax of CML. The semantic function is specified by a set of translation rules that apply to particular constructs of SysML.
## Timetable:

<table>
<thead>
<tr>
<th>Event</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>First version for contributor’s review</td>
<td>01.10.12</td>
</tr>
<tr>
<td>Second version for contributor’s review</td>
<td>01.01.13</td>
</tr>
<tr>
<td>Final version for internal review</td>
<td>08.03.13</td>
</tr>
<tr>
<td>Reviewer’s comments</td>
<td>22.03.13</td>
</tr>
<tr>
<td>Final version for submission</td>
<td>29.03.13</td>
</tr>
</tbody>
</table>
## Contents

1 Introduction ................................................. 9  
   1.1 Motivation ............................................. 9  
   1.2 Objectives ............................................ 12  

2 Preliminaries ................................................ 14  
   2.1 CML ................................................... 14  
   2.2 SysML .................................................. 20  
   2.3 Example ............................................... 30  
   2.4 Final remarks .......................................... 36  

3 Formal semantics of SysML models ......................... 37  
   3.1 Guidelines ............................................ 37  
   3.2 Semantic rules metalanguage .......................... 42  
   3.3 Overview of the semantics of SysML models ......... 44  
   3.4 Final remarks .......................................... 47  

4 Structural diagrams ........................................... 49  
   4.1 Overview ............................................... 50  
   4.2 Operations and signals ................................. 53  
   4.3 Interfaces ............................................. 55  
   4.4 Standard ports ......................................... 57  
   4.5 Simple blocks .......................................... 61  
      4.5.1 Simple blocks: types ............................. 61  
      4.5.2 Simple blocks: channels ......................... 64  
      4.5.3 Simple blocks: processes ....................... 65  
   4.6 Composite blocks ...................................... 72  
   4.7 Final remarks .......................................... 77  

5 State machine diagrams ....................................... 79  
   5.1 Examples ............................................... 81  
   5.2 Formal semantics ....................................... 86
Chapter 1

Introduction

In this report, we propose a formal semantics of SysML based on the CML notation. We identify a set of guidelines of usage for SysML diagrams that support the definition of a self-contained semantics of SysML models. Additionally, we identify a subset of diagram constructs that are more frequently used.

Based on these guidelines, the semantics of SysML is given as a translation from the abstract syntax of SysML to the abstract syntax of CML. Our translation is formalised by semantic functions that can also be regarded to define a refinement semantics for SysML diagrams using CML. The functions are defined by translation rules, and are inductive on the structure of SysML models.

In particular, we address the translation to CML of model that include the following diagrams: block definition diagrams, internal block diagrams, activity diagrams, sequence diagrams, and state machine diagrams. The translations are performed by applying translation rules according to a pre-defined strategy. The parts of the model represented by the different diagrams are translated by following specific strategies. All translation strategies are systematic and mechanisable.

In the next sections, we discuss our motivations (Section 1.1) and give an overview (Section 1.2) of the approach taken to linking SysML and CML.

1.1 Motivation

The design of Systems of Systems (SoS) is a field of study that presents major challenges due to their complexity. Traditional approaches lack in support for analysis during the development process, which leads to inefficiencies in cost,
time and design. COMPASS (Comprehensive Modelling for Advanced Systems of Systems) aims at extending the industrial practise by means of a set of tools and techniques that support a formally grounded model-based approach for the development of SoS.

In this context, the work reported in this document is directly related to the link between the industrial practice and the formal world. By linking a current industrial notation (SysML) with a formal notation (CML), a new realm of tools and techniques become available to SysML. Informal development can be carried out with the additional guidance of the analysis techniques supported by CML, and, when needed, formal development can be introduced.

SysML diagrams are intended to describe views on an underlying model that ensures informally the consistency between the model elements contained in those diagrams. It is not possible, however, to give a (useful) formal semantics to an arbitrary collection of SysML diagrams. They do have a standardised syntax and clearly identified well formedness rules [Obj10a]. However, SysML places no restriction on the semantics of some its elements. For example, there is no restriction on the meaning of a block in a block definition diagram (BDD), nor on what it may represent. The diagrams of SysML may be used in several ways, and their use is dependent on the subject being modelled. We must, however, ensure formally that the diagrams are used in a consistent manner. Therefore, we propose guidelines that restrict the usage of the diagrams and define the semantics of SysML for that particular usage, so that a useful CML model can be produced.

Modelling patterns, such as those prescribed by Holt et al. [HP08, Hol09], provide a systematic method for producing architectural models. For example, Holt defines a pattern for business process modelling that consists of a collection of views in the form of SysML diagrams, describes the consistency rules between those views, and a method for defining diagrams in a systematic manner [Hol09]. An initial architectural modelling pattern for requirements engineering is given by Holt et al. [HPM11], and is expanded to cover SoSs in [HPHH12]. One of the objectives of the COMPASS project is to define patterns for SoS modelling. Therefore, we aim to provide a method to be used by SoS engineers for defining a SysML architectural model containing diagrams required for deriving CML models. Such patterns define indirectly a semantics for the model elements of the SysML diagrams, and their relationship. It is this informal semantics that we can capture formally in CML.

Such patterns are currently under development and discussion in COMPASS. Work package WP21 will provide an approach to defining architectural models for SoSs using architectural patterns that inform the methods for developing SoS architectural models in SysML. Figure [1.1] provides an overview of the diagram
types of use and how they may relate to CML models. The work of task T2.2.2 (including the work documented in this deliverable) will inform the choice of diagrams and the information that needs to be conveyed in those diagrams.

At this stage, we define the semantics of diagrams whose patterns of usage are already well understood, namely, block definition diagrams, internal block diagrams, state machine diagrams, the sequence diagrams, and the activity diagrams. These diagrams have already been extensively discussed in the literature, both inside and outside the context of SysML. The definition of the semantics of these structural and behavioural diagrams is the subject of this deliverable. We propose a general approach to modelling diagrams using parallelism, and go well beyond what is available in the literature by covering the connections between the semantics of the individual diagrams. In this way, we produce a CML model of the combined views portrayed in the individual diagrams.

Whilst SysML is a graphical and largely informal notation for the specification of structural and behavioural aspects of systems, CML is a formal textual nota-
tion that supports the specification of systems in different levels of abstraction. Additionally, CML supports the specification of contracts, which is not directly supported in SysML.

CML supports the specification of both state-rich behaviour and communication. The two central notions in CML are processes and refinement: processes encapsulate behaviour and refinement supports the comparison of processes. A process may be defined in terms of other processes or in terms of an encapsulated state and actions that potentially modify the state. The actions are specified through a mixture of CSP and VDM operators. The first set of operators focus on communication (e.g., prefixing) and control structure (e.g., interleaving), whilst the second set of operators focus on data aspects (e.g., assignment). The values manipulated by data operations in CML are drawn from the types traditionally found in specification languages (e.g., sets, sequences) as well as from classes. A class in CML declares a state and a number of operations that manipulate the state. Operations in CML are purely state operations, as opposed to actions, which support the full range of CML action constructors.

Additionally, CML supports step wise development by means of algebraic refinement laws. The soundness of the refinement laws is established with respect to the formal semantics of CML, defined in the Unifying Theories of Programming [HJ98]. CML is still under development, with a COMPASS tool and several analysis plug-ins currently in production [CML+12]. In particular, tool support for CML will include a parser, a type-checker, a simulator, a theorem prover, a model-checker and a refinement editor.

### 1.2 Objectives

In this document, we aim at setting the approach to realise the link between SysML and CML. We aim to identify those diagrams that provide details relevant to CML. As identified by Holt et al. [HNH+12], SysML has two main types of diagram: structural and behavioural. The structural diagrams of SysML consist of block definition diagrams, internal block diagrams, package diagrams, and parametric diagrams. Behavioural diagrams include activity diagrams, sequence diagrams, state machine diagrams and use case diagrams. SysML also includes a new requirements diagram. For the basis of linking SysML and CML, we initially define the semantics of the following diagrams: block definition diagrams, internal block diagrams, activity diagrams, sequence diagrams and state machine diagrams.
The approach adopted for the combination of SysML and CML takes into account elements of SysML that can be systematically formalised (e.g., state machine diagrams), elements whose formalisation depends on particular uses (e.g., block diagrams), and elements that cannot be fully formalised (e.g., requirements diagrams). Our approach consists of deriving CML processes for the SysML behavioural diagrams, deriving CML processes and classes from SysML structural diagrams, and compose the derived processes in parallel to yield the model of the whole system. Finally, links between the CML model and the non-formalised elements of SysML can be established.

Figure 1.1 presents an overview of the approach. Arrows from SysML diagrams (e.g., state machine diagrams) to CML processes represent the translation of such diagrams, and arrows between CML processes represent process composition.

Block definition diagrams and internal block diagrams are used to derive types and classes, which are encapsulated in processes that offer as services the attributes of the block as well as its operations via channels. Activity diagrams can describe the body of the operations of the blocks. State machine diagrams may restrict the behaviour of blocks by enforcing specific protocols regarding the use of operations and attributes. Finally, sequence diagrams generate processes that model particular scenarios and can be used to validate the behaviour of the system using traces refinement.

The processes derived from activity, sequence and state machine diagrams can be generated by means of a set of translation rules. The models generated by these translation rules form the basis of the link between SysML and CML.

This report is structured as follows. In Chapter 2 we introduce both CML and SysML. Chapter 3 presents the set of guidelines a SysML model must follow to be given a semantics, the notation used in the exposition of the translation rules, and the translation rule for SysML models, which builds on the rules presented in the remaining chapters. Chapter 4 describes the translation to CML of block definition diagrams and internal block diagrams. Chapters 5, 6 and 7 present the formalisation in CML of state machine diagrams, sequence diagrams and activity diagrams, respectively. Chapter 8 concludes and discusses future work.
Chapter 2

Preliminaries

In this chapter, we present the background material necessary for the understanding of the proposed formal semantics. In Sections 2.1 and 2.2, we introduce the two notations central to our work, namely, CML and SysML. Section 2.3 presents a simple SysML model that uses all diagrams covered in this document.

2.1 CML

The COMPASS modelling language (CML) \cite{WCF12} is a formal specification language that integrates a state based notation (VDM++ \cite{FL09}) and a process algebraic notation (CSP \cite{Hoa85}), as well as Dijkstra’s language of guarded commands \cite{Dij75} and the refinement calculus \cite{BvW98}. It supports the specification and analysis of state-rich distributed specifications.

We introduce CML by means of a specification of a simple clock. For more details, we refer to \cite{WCF12,WCC12}.

A CML specification consists of a number of paragraphs, which at the top level can declare types, classes, functions, values (i.e., constants), channels, channel sets, and processes. Both classes and processes declare state components, and may contain paragraphs declaring types, values, functions and operations. Processes may additionally declare any number of actions, and must contain an anonymous main action, which specifies the behaviour of the process.

Initially, we specify a simple clock whose only observable behaviour is a synchronisation on a channel \texttt{tick} used to mark the passage of seconds.

\begin{verbatim}
channels
\end{verbatim}
As previously mentioned, a class declares a state and a number of operations that
manipulate the state. In particular, a class may declare an initialisation operation
identified by the keyword initial, which acts as a constructor of the class.

Internally, the clock has a state variable \( s \) that records the number of seconds that
have elapsed, and has two operations defined: Init() and increment. The first
simply initialises the state with 0, and the second adds one to the state component.
The state is captured by the following class declaration.

```plaintext
class ClockSt = begin
  state
    public s: nat
  initial
    public Init()
    frame wr s
    post s = 0
  operations
    public increment()
    frame wr s
    post s = s~ + 1
end
```

The frame keyword in the declaration of operations specifies the state components
that can be read and written by the operation. In the case of the Init operation,
the state component \( s \) can be written by Init. The post keyword specifies the
post condition of the operation. In the case of Init, the post condition states that
the state component \( s \) (after the operation) is equal to zero. The post condition of
the operation increment equates the state component \( s \), after the operation, to the
sum of its initial value (\( s~ \)) and one.

The class ClockSt is used in the definition of the process that specifies the be-
behaviour of our clock. Similarly to a class, a process encapsulates a state and may
contain operations, but in addition it contains at least one action (the main action),
which specifies the behaviour of a process.

Our simple clock initialises its state, waits for one time unit, which we take to
mean one second, increments its counter and synchronises on tick. This is spec-
ified by the following process declaration.

```plaintext
process SimpleClock = begin
  state c: ClockSt
  actions
    Ticking = Wait 1; c.increment(); tick -> Skip
```
The simple clock is a process that declares a state and a number of actions. The state, in this case, is formed by a single state component \( c \) of type \( \text{ClockSt} \). The actions include \( \text{Ticking} \) and the action started by \( @ \). It simply initialises the state by calling the operation \( \text{Init()} \) of the state component \( c \) and recursively (\( \mu \)) calls the action \( \text{Ticking} \). This action waits for one time unit (\( \text{Wait 1} \)), increments the internal counter and synchronises on the channel \( \text{tick} \).

Our initial specification of the clock is extremely simple; the only observable event is the synchronisation on \( \text{tick} \). It might be interesting to have a clock that takes advantage of its internal counter and supplies information about how many seconds, minutes, hours and days have elapsed.

We now extend our simple clock to include this additional functionality. First, we declare four additional channels that communicate a natural number. They are used to query the seconds, minutes, hours and days that have elapsed.

```plaintext
channels
  second, minute, hour, day: nat
```

The new clock specification is similar to the simple clock; it declares the state of the process as containing just the component \( c \) of type \( \text{ClockSt} \), but additionally defines three functions: \( \text{get_minute} \), \( \text{get_hour} \) and \( \text{get_day} \). They take the number of seconds recorded in the state, and calculate, respectively, the equivalent number of minutes, hours and days.

```plaintext
process Clock = begin
  state c: ClockSt
  functions
    get_minute(s: nat) m: nat
    post m = s/60

    get_hour(s: nat) h: nat
    post h = get_minute(s)/60

    get_day(s: nat) d: nat
    post d = get_hour(s)/24

  actions
    Ticking = Wait 1; c.increment(); tick -> Skip
```

The ticking action remains the same as before, but we add a new action, \( \text{Interface} \), that provides the extra functionality.
This action simply offers an external choice (||) of communication over the channels second, minute, hour and day, and recurses. Each communication outputs (indicated by ! after a channel name) the appropriate value calculated using the functions previously defined.

Now, the main action of the new clock is slightly different. It first initialises the state as before, but instead of offering Ticking alone, it composes Ticking in parallel with the recursive action Interface with the option of interrupting (/\) Interface with a synchronisation on tick. The operator A1[||ns1| cs |ns2||]A2 composes the actions A1 and A2 in parallel based on a set of events cs on which the two parallel actions synchronise, and two name sets ns1 and ns2 that partition the state of the process and indicate which state components can be updated by the left (ns1) and right (ns2) parallel actions. In our example, the action Ticking can update the state component c and the right parallel action does not update the state. The parallel actions synchronise on the channel tick.

@ c.Init(); mu X @ {
    Ticking [||{c}|{||tick||}|{}|] (Interface/\tick -> Skip)
}; X
end

While Ticking is waiting, the right hand side of the parallelism can offer any number of interactions over the channels in Interface. When Ticking finishes waiting, s is incremented, and the parallelism synchronises on tick. In this case, the action Interface is interrupted and both sides of the parallelism terminate. At this point, the recursive call (on X) takes place.

When the parallelism starts, both parallel actions take a copy of the state, and when the parallelism terminates, the state is updated based on the changes performed by the two parallel actions (on their copies of the state) and the partition of the state. A consequence of this is that changes to the state performed by Ticking can only be reflected on the behaviour of Interface when the parallelism terminates, the state is updated and Interface restarts (as part of the recursive call) with a copy of the updated state.

Now we have a clock that not only signals the passing of time, but can also output the time. However, we might also want to be able to restart the clock. For this, we
define a channel \texttt{restart} and a new clock \texttt{RestartableClock}.

channels
restart

The definition of the restartable clock is similar to that of the process \texttt{Clock} defined above. The process \texttt{RestartableClock} has a new action \texttt{Cycle}, and its main action offers the action \texttt{Cycle} and the possibility of interrupting it through the channel \texttt{restart}. If the interruption takes place, the main action recurses and \texttt{Cycle} is called resetting the state.

\begin{verbatim}
process RestartableClock = begin
  state c: ClockSt
  functions
    get_minute(s: nat) m: nat
    post m = s/60
    get_hour(s: nat) h: nat
    post h = get_minute(s)/60
    get_day(s: nat) d: nat
    post d = get_hour(s)/24
  actions
    Ticking = Wait 1; c.increment(); tick -> Skip
    Interface = second!(c.s) -> Interface
    minute!(get_minute(c.s)) -> Interface
    hour!(get_hour(c.s)) -> Interface
    day!(get_day(c.s)) -> Interface
    Cycle = c.Init(); mu X @ {
      Ticking [||c||{tick}||{}||] (Interface/\tick -> \texttt{Skip})
    };
    @ mu X @ Cycle /\ restart -> X
end
\end{verbatim}

We can further extend the functionality of the clock by specifying a multi-clock. A simple way of defining such a clock is to compose a number of restartable clocks (or any other variety of clock). This raises the question of how the clocks are composed. For instance, do all clocks synchronise on tick? Can they be restarted on a one by one basis? We present below two processes that model a multi-clock. Both of them assume that the clocks are synchronous, but the first allows independent restarting, while the second does not.

First, we define the channels that allow the environment to communicate with
specific clocks. We assume that the clocks in the multi-clock are indexed using natural numbers, and are similar to those already defined, communicating a natural number (the identifier of the clock) and the value originally communicated. We prefix the name of the new channels with an \( i \).

channels
- isec, iminute, ihour,iday: nat \* nat
  irestart: nat

Our first model of a multi-clock is specified by the process \( \text{NRestartableClocks}_1 \). This is a parametrised process that takes the number \( n \) of clocks, and starts \( n \) copies of \( \text{RestartableClock} \) running in parallel and synchronising on \( \text{tick} \). The channels in the \( \text{RestartableClock} \) process need to be renamed to distinguish one clock from another. We rename each channel (except \( \text{tick} \)) to its \( i \) version.

\[
\text{process} \ \text{NRestartableClocks}_1 = n: \text{nat} @
\quad \{ ||\{\text{tick}\}|| \} \ i: \{1,\ldots,n\} @
\quad \text{RestartableClock}[[\text{second} <- \text{isecond}.i,
\quad \text{minute} <- \text{iminute}.i,
\quad \text{hour} <- \text{ihour}.i,
\quad \text{day} <- \text{iday}.i,
\quad \text{restart} <- \text{irestart}.i]]
\]

Our alternative process \( \text{NRestartableClocks}_2 \) is similar, except that the different clocks synchronise on \( \text{restart} \) as well, and this channel is not renamed. Thus, a synchronisation on \( \text{restart} \) restarts all the clocks simultaneously.

\[
\text{process} \ \text{NRestartableClocks}_2 = n: \text{nat} @
\quad \{ ||\{\text{tick, restart}\}|| \} \ i: \{1,\ldots,n\} @
\quad \text{RestartableClock}[[\text{second} <- \text{isecond}.i,
\quad \text{minute} <- \text{iminute}.i,
\quad \text{hour} <- \text{ihour}.i,
\quad \text{day} <- \text{iday}.i]]
\]

One might consider that, whilst these definitions are reasonably intuitive, they are not the most efficient for implementation purposes. So, one might implement a multi-clock with a single clock that provides all services on the \( i \) channels. To model this design, we simply associate each channel of a restartable clock with the equivalent \( i \) channel, but ranging over all the possible clocks.

\[
\text{process} \ \text{NRestartableClocks}_\text{Impl} = n: \text{nat} @
\quad \text{RestartableClock}[[\text{second} <- \text{isecond}.i,
\quad \text{minute} <- \text{iminute}.i,
\quad \text{hour} <- \text{ihour}.i,
\quad \text{day} <- \text{iday}.i | i \ \text{in set} \ \{1,\ldots,n\}]]
\]
This process simply renames each channel of RestartableClock (except tick and restart) to a set of communications on the associated i channel communicating the identifiers of the clocks.

This process raises the question of which of our multi-clock processes is being implemented by NRestartableClocksImpl. These questions can be formulated as follows.

assert NRestartableClocks1 [= NRestartableClocksImpl
assert NRestartableClocks2 [= NRestartableClocksImpl

The first assertion states that the process NRestartableClocksImpl is a refinement of the process NRestartableClocks1, and the second asserts that the implementation is a refinement of NRestartableClocks2. For some models, this assertions can be checked using a model-checker, but for others, a theorem-prover may be necessary. The CML tools will help answer such questions.

More details about the CML notation will be presented as needed in the sequel.

2.2 SysML

SysML, standardised by the Object Management Group [Obj10a], is a modelling notation for systems engineering, defined as a UML 2.0 profile. SysML allows the representation of systems, hardware, software, information and processes, with the objective of providing dedicated support for system-level modelling, verification and validation. Like UML, SysML provides a number of diagrams to support the description of different aspects of a system. SysML has the following prominent distinctive features that are not present in UML:

- The "classical" software-centric focus present in UML, through class diagrams and composite structure diagrams, has been moved to the system-level in SysML by the introduction of block definition diagrams and internal block diagrams [Obj10a, Chapter 8].

- The UML notion of interfaces has been focused in SysML on system-level interfaces by the introduction of service-based and flow-based ports and interfaces [Obj10a, Chapter 9].

- The general UML notion of constraints has been strengthened in SysML through the introduction of constraint blocks and parametric diagrams [Obj10a, Chapter 10].
• SysML introduces a notation for requirements engineering and tracing from model elements to requirements [Obj10a, Chapter 16].

An overview of SysML, in the context of SoS architectural modelling in COMPASS, is provided by Holt et al. [HNH+12]. In this section, we shall briefly describe the parts of SysML structural and behavioural diagrams that are relevant in the context of the translation to CML.

**Block Definition Diagram**

A block definition diagram depicts blocks and how they relate to each other. A block is a very general modelling concept that represents any abstract or real entity. For instance, a block can be a piece of hardware, a software, a physical object, an abstract entity, or a system. A block is represented as a rectangle. Figure 2.1 shows an example of a block definition diagram used in the model of an air compressor system in [FMS11]. We can see seven blocks in Figure 2.1 representing a Compressor, an Air Compressor, an Air Tool, a Motor Controller, a Motor, a Pump, and a Tank.

![Block Definition Diagram](image)

Figure 2.1: A block definition diagram of an air compressor [FMS11].

Blocks can have attributes and operations. Attributes are properties of a block. For instance, volume is an attribute of the Tank block (Figure 2.1). An operation is something that the block does [HP08]. For instance, the Tank is able to inform its users the pressure inside itself (operation `get_pressure()` in Figure 2.1). Operations are typically triggered by synchronous requests [FMS11]. Receptions are operations triggered by asynchronous requests and are associated with signals. Attributes, operations, and receptions are declared in different compartments inside the block.
We define that a block is related to another block by connecting them with an association. Associations are represented graphically as a solid line connecting the blocks. Associations can have names and can relate blocks over different multiplicities. For instance, the Air Compressor block is associated with one Air Tool block in Figure 2.1.

A different way to relate blocks is via the aggregation operator. The aggregation says that a block is made up of other blocks [HP08]. The aggregation is depicted as diamond in block definition diagrams. Figure 2.1 shows that an Air Compressor is made up of a Motor Controller, a Motor, a Pump, and a Tank.

Composition is a whole-part relation in which the parts cannot exist by themselves, as it is an inherent part of the whole. For instance, a Student is composed of a block Schedule. If the Student is removed, so is Schedule. The lifetime of the parts is dependent on the lifetime of the whole. Composition is represented by a filled diamond.

We can define ports for a block, which are interaction points from which a block communicates to its environment. A port is usually associated with an interface, which can be either provided or required. If a block takes inputs from an interface and acts on that input, that interface is a provided interface [HP08]. If a block generates outputs to be used by another block, that is a required interface [HP08]. Ports are depicted as a small rectangles attached to the edges of a block. Required interfaces resemble semi-circles, while provided interfaces are represented by circles. Figure 2.1 shows that the Motor Controller provides an interface to a Sensor and requires an interface from an Actuator.

Blocks can be declared as being a particular type of another block. For instance, the Air Compressor is defined as a type of Compressor. Compressor in Figure 2.1 has all the features shared by any compressor. And Air Compressor just inherits those features. Generalisations like this are represented graphically by an arrow head pointing from the specialised block towards the general block.

**Internal Block Diagram**

Internal block diagrams are very similar to block definition diagrams. They model the internal structure of a block and typically show the internal connections between parts of a block [FMS11, HP08]. Figure 2.2 shows the interconnections of the components of the Air Compressor. We have omitted the interface symbols for simplicity. Some of the ports depicted in Figure 2.2 are not standard, but are flow ports, i.e. interaction points where material, data or energy flows into or goes out of a block.
State Machine Diagram

While SysML state machine diagrams are syntactically more restricted than UML state machine diagrams, their behaviour conforms to the behavioural semantics of the latter [Obj10b, pp. 541]. A state machine diagram reacts to events from the environment, which are stored in an event pool. The order in which events in the event pool are processed is unspecified. Figure 2.3 shows an example of a state machine.

Figure 2.3: Monitored machine: state machine diagram

Simple states are states that do not have substates (e.g., state off in Figure 2.3). States with substates are called composite (e.g., state on in Figure 2.3) and have
one or more regions (e.g., region \textbf{R1} in state \textbf{on} in Figure \ref{fig:state}) that group its substates. A composite state with two or more regions is called an orthogonal state, and we refer to composite states with a single region as non-orthogonal states. Multiple regions of a composite state are visualized by dashed lines that partition the inside of the state, and all regions of a state are active when the state is active, and vice-versa. Each region contains a number of substates, and when the region is active, only one of its substates can be active.

State may contain three types of behaviour: entry and exit actions, and “do activities” (e.g., the activity \texttt{check1} in Figure \ref{fig:state}). Entry actions are executed when the state is entered, exit actions are executed when the state is exited, and “do activities” are executed when the state becomes active. “Do activities” may continue indefinitely and may be interrupted by a transition that exits the state.

When entering a non-orthogonal composite state or a region, the decision of which substate to enter can be specified by initial pseudostates or history pseudostates. Initial pseudostates (represented graphically by solid black circles as shown in Figure \ref{fig:state}) provide a transition path (that is, a sequence of transitions connected by pseudo state) to the state that must be entered.

A history pseudostate records the most recently entered substate of the state or region that contains it. When the state or region containing the history pseudostate is entered, the last active substate is entered; it is represented graphically by a circle with a capital letter H inside. There are two variants of history pseudostates, shallow history, which affects only the state or region containing the history pseudostate, and deep history, which affects the state or region containing it and all of its substates and subregions. Deep history pseudostates are represented as shallow history pseudostates with an asterisk next to the H.

The connection between the initial pseudostate and the substates of its parent is specified by transitions. In general, transitions connect source and target vertices (states or pseudostates), and may contain a trigger, guard and behaviour. The transition from the state \texttt{off} to the state \texttt{on} in Figure \ref{fig:state} is triggered by a signal \texttt{off}, and has a behaviour \texttt{pl.print("Turning off")}. For the purpose of this work we divide transitions between completion and non-completion transitions. Completion transitions are triggerless transitions leaving a state that are executed when the internal behaviour of the state (“do activities” and substates) terminates. Non-completion transitions are triggered by events from the event pool and can interrupt the internal events of the its source states.

A transition between two states, whose trigger and guard (if any) are evaluated to true, can be executed (provided it does not conflict with transitions with higher priorities) by exiting the source state, executing the transition behaviour and en-
tering the target state. In the general case, transitions can cross state boundaries; this requires that not only the source state is exited by the transition, but some of its ancestors too. The highest level states that must be exited as a result of a transition are the subtstates of the least common ancestor of the source and destination of the transition.

Two or more transitions can be connected by junction pseudostates (represented graphically by a solid black circle) to form a transition path that essentially behaves as a transition with multiple source and target states. A junction pseudostate provides a mechanism for choosing which transition (leaving the junction pseudostate) to follow. Transitions leaving junction pseudostates cannot have triggers, and the evaluation of the guard is performed prior to the execution of the transition. An alternative is the choice pseudostate that provides a mechanism for choosing the particular transitions to follow, but, unlike junction pseudostates, the decision of which transition to follow is performed during the execution of the transition path. Therefore, the execution of the behaviours of the transitions in the path can affect the outcome of the execution of the choice pseudostate.

A non-orthogonal composite state (that is, a composite state with a single region) or a region may contain a final state. When reached, the final state indicates that the internal behaviour of the composite state or region has terminated. In conjunction with the termination of the “do activities”, this can trigger the execution of completion transitions.

Finally, state machines may contain join and fork pseudostates. Join pseudostates (e.g., right vertical bar in Figure 2.3) provide a way of gathering transitions leaving regions of an orthogonal composite state and joining them into one single transition targeting a state (outside the composite state). The transitions associated with a join pseudostate are enabled when all transitions reaching it are. When fired, all the source states are exited, and the target state is entered. A fork pseudostate (e.g., left vertical bar in Figure 2.3) behaves in the opposite direction; it links a single state to states in orthogonal regions. When the transition reaching the fork pseudostate is executed, all target states are entered.

**Sequence Diagram**

In UML 2.0, there are four types of diagram to describe interactions: sequence diagrams, communications diagrams, interaction overview diagrams, and timing diagrams. Among them, the sequence diagram is the most commonly used to describe interaction aspects of systems, and therefore SysML considers only sequence diagrams to describe interactions. According to the SysML specifica-
tion [Obj10a], communications diagram and interaction overview diagrams are excluded as they overlap in functionality without adding significant capabilities for modelling systems. Timing diagrams are not included because of its maturity and suitability for Systems Engineering needs.

The main purpose of sequence diagrams is to describe operational scenarios of a system with an emphasis on time [HP08]. This concept is introduced through the use of an element known as a lifeline. Each participant of the diagram (typically a block) possesses a lifeline where we can represent a message exchange ordering.

Messages can be of three types: asynchronous (open arrow), call (synchronous - closed arrow), or reply from a call (dashed arrow). They are used to represent a communication from one participant to another. SysML also has available a collection of operators like the parallel composition, the conditional, the loop, etc. called combined fragments.

Figure 2.4 presents an example in [HP08] where the scenario being described is the movement of a robot. There are four lifelines (Sensor, Controller Software, Motor A and Motor C), nine messages exchanged between them, and two PAR combined fragments denoting parallel behaviour. For example, sensorHit() is an asynchronous message going from the Sensor to the Controller Software, and turn() is a synchronous message going from the Controller Software to itself. The Sensor sends a message to the Controller Software, which in turn, sends a turn() message to itself. Then, the Controller Software sends two messages in sequence to Motor A and two messages in sequence to Motor C. This transmission happens in parallel (see the PAR combined fragment on top). Finally, the message move() is sent from the Controller Software to itself, which in turn, sends two messages in parallel: one to Motor A, and another to Motor C.

Whilst SysML sequence diagrams have a rich syntax, in Chapter 6, we detail the key elements relating to the initial set of translation rules.

**Activity Diagram**

The SysML activity diagram is imported from UML 2.0 with some extensions. This type of diagram is based on classic flow charts. According to Holt and Perry [HP08], activity diagrams can be used for low-level modelling in comparison to the previous diagrams. Their main utility is to detail the behaviour of an operation, although other uses include describing workflows or processes.

An activity diagram has three basic elements: activity nodes, edges and regions.
An activity node can represent an action, a control or an object node. An action node is used to represent some action, for example, *Turn Key To On* or *Press Gas*. Control nodes are used to manipulate the flow of actions, for example via decisions, forks and joins. Object nodes are blocks defined elsewhere that are inputs or outputs of an activity. Edges can be of two types: control or object flow. The former shows the possible routes through activities and the latter details the data flow among them. Regions may be used to describe an activity that can be interrupted or to describe partitions of the diagram, also known as *swimlanes*.

Activity diagrams (ADs) are modelled basically to represent flow-based behaviour. Object flows describe how input and output items flow between actions. These can be either continuous, like water from a pump, or discrete, like data from a system. Control flows provide information about the possible routes for the execution of actions, that is, when and in which order the actions will run. Control nodes help to specify the organisation of such flows.

The graphical representation of an activity diagram is designed as a rectangular frame around the diagram itself with a name in a compartment in the upper left corner (Figure 2.5). An action is an elementary behaviour of the activity. The
composition of actions builds the activity of a diagram. Figure 2.6 shows the graphical representation of an action. Object nodes (Figure 2.7) are instances of a particular item (e.g. block or part) already existent in other diagrams. They may be available at a particular point in the activity and can be represented by rectangles containing information on the block type and state or can connect with actions through the use of pins. A pin is represented by a small rectangle and usually contains an arrow to show whether it is an input or an output pin. Pins can carry information on the type of the item being transmitted. Control nodes are used to coordinate the flows between other nodes. Figure 2.8 illustrates the possible control nodes. We describe each of them in what follows.

A decision node splits the flow according to some guards or probabilities. A decision node has just one incoming edge. A merge node brings together multiple alternate flows without synchronisation. A fork node splits a flow into multiple concurrent flows. A join node synchronises multiple flows. An initial node starts the flow when the activity is invoked. An activity may have more than one initial
node, then invoking an activity starts multiple flows. An activity may have more than one activity final node. The first final node reached stops all flows in the activity. A flow final node destroys all tokens that arrive at it. It has no effect on other flows in the activity.

An activity edge may be of two types: control flow or object flow. The former controls the order in which the actions may occur and no information of an object node pass through a control flow edge. The latter is the opposite, as it is used to represent the flow of information through activity nodes. Figure 2.9 shows the graphical syntax of control flows (Figure 2.9(a)) and object flows (Figure 2.9(b)). An accept event action is an action that waits for the occurrence of an event meeting a specified condition. Figure 2.10 displays the representation of an accept event action. A send signal action creates a signal instance from its inputs, and transmits it to the target object, starting the execution of an activity. Send signal actions are represented by a convex pentagon as shown in Figure 2.11.

In the next section, we present an example that illustrates the use of all diagrams we have described above.
2.3 Example

To illustrate the use of multiple diagrams in a SysML model, we present a simple example of a system formed by a machine, two monitors and a screen. This model provides an abstract view of a system with failure monitoring. The machine can be on, off or in a failure state, and can receive requests to turn on or off, as well as an indication that it has been fixed. The two monitors, when prompted to do so, evaluate the machine and if they observe a fault, the fault is reported. Once activated, the machine iteratively requests both monitors to check for failures, and it only goes into a failure state if both monitors agree that a failure occurred. All requests and changes of machine state are reported in the screen.

Our focus on this example is the machine and how it uses the monitors to move to a failure state if necessary. For this reason, the main behaviour of the machine is left unspecified, as well as the details of how the monitors check the machine for failures and how information is printed in the screen. The check and print functionalities of the monitors and screen are modelled as operations, but no model of the behaviour is provided, since our interest in this model is the occurrences of calls to these operations and not their behaviours.

The bdd for the system is shown in Figure 2.12. The block System has four composition relations, all of which have cardinality 1. The labels in the composition relations indicate the role played by each part, that is, machine, screen, monitor1 and monitor2. The block monitor has a single operation check with an output parameter b. It returns false if it observes a failure in the machine, and true otherwise.

The bdd does not provide any information about the interconnection between the different parts of the system. This is described in the internal block diagram of the block System. This diagram is shown in Figure 2.13.

The ibd shows the block System as the outermost block, and the instances of its parts inside the block. There are exactly four block instances named after the role names shown in the composition relation in the bdd. The block System has a port, p3, that provides an interface I (indicated by the circle at the end of the line
labelled with the interface) to the external world. The interface that is provided by the port to the external environment is required from the internal environment. In Figure 2.13 the counterpart of the provided interface $I$ on the port $p_3$ is shown as a required interface (line with a semicircle at the end) facing the inside of the block.

We observe that some of the information captured in the SysML model is not shown in diagrams (like the signals in interface $I$, for example), but is captured as properties of the elements of the diagram. The contents of the interface (although not shown in the diagram) can be recovered by inspecting the model. The three interfaces shown in Figure 2.13 are defined as follows. The interface $I$ contains three signals: $on$, $off$ and $fix$. The interface $IO$ defines the signature of the operation $print$ that takes a string as an input parameter. Finally, the interface $I$ defines the signature of the operation $check$ that has an output parameter $b$ of type $bool$.

The port $p_3$ is connected to the port $p_1$ of the part $machine$; this indicates that requests received at the port $p_3$ are relayed to the port $p_1$. Additionally, messages
Figure 2.13: Monitored machine: internal block diagram

from the port $p_1$ can be directed at the port $p_3$ and can be available to the external world. Since there are two connections between the port $p_1$ of machine and other ports, the actual relaying of messages depends on the particular message. For instance, messages of the interface $I$ are only sent to the port $p_3$, and messages of the interface $IO$ are only sent to the port $p_2$ of screen. If the message is allowed by both interfaces, it is sent non-deterministically to one of the possible targets.

The part machine has three ports: $m_1$, $m_2$ and $p_1$. The first two are connected to the parts monitor1 and monitor2 respectively, and the third is connected to the port $p_3$ of the block System as previously mentioned. The two instances of Monitor are connected to the ports of machine through a port $p$ and the screen is connected to machine through its port $p_2$. These are the only means of communication between the different parts of the system.

The behaviour of the monitors and the screen are simple and are left underspecified, whilst we focus on the behaviour of the block Machine. The behaviour of this block is specified by the state machine diagram shown in Figure 2.14.

Initially, an instance of Machine is in the state off, and upon receipt of an event signal on, it sends a request to print “Turning on” to the port $p_1$, deactivates the state off and enters the state on. The state on is entered through a transition path involving a fork pseudostate, which directly activates the states monitor of
the two regions (R1 and R2) of the state on. Before the states reached by the fork pseudostate are activated, however, all its ancestors (on, R1 and R2) are first activated. Once the two monitor states are entered the “do activities” start executing, and if both of them terminate the completion transitions from the states monitor to the join pseudostate takes place. In this case, the machine exits the state on and all its substates, sends a request to print to the port p1 and enters the state fail. If a signal event off is received before both do-activities terminate, they are interrupted, the state on (and its substates) exit, a print request is sent to port p1 and the state off is entered. Once in the state fail, the machine can only be restarted after a signal event fix has been received. In this case, the state fail exits, a print request is sent to p1 and the state off is entered.

The “do activities” of the states monitor are check1 and check2 and are specified by activity diagrams. The diagram for activity check1 is shown in Figure 2.15. This activity initially obtains a reference to the block that calls the activity, obtains the port m1 of the block, and call the operation check on it. If the result (in pin b) is true, the activity iterates, otherwise it terminates. The diagram for the activity check2 is similar.

Finally, we use sequence diagrams to describe possible (or undesirable) executions of the system. For instance, the diagram in Figure 2.16 shows one possible execution of the system. An external user sends a signal event on to the machine,

---

1Sequence diagrams specifying positive and negative scenarios can be distinguished by an appropriate naming convention or package structure.
which prints some information to the screen, and in parallel requests the operation check of both monitors. In this scenario, the first monitor detects a failure in the first call, and the second monitor takes two iterations to recognize a failure. At this point the machine prints to the screen a failure warning. The user then sends a signal `fix`, the machine prints a message to the screen and switches off.
We also specify a scenario that should not be allowed by our systems. This is shown in Figure 2.17. Up to the end of the do-activities (parallel box), this scenario is similar to the previous one, however, once the activities terminate, no print request is sent to the screen, and immediately, the fix signal is received. This should not be allowed by the system. Additionally, after the machine is fixed, and upon receipt of a signal off, the machine sends a print request to the screen. This communication pattern should also not be allowed by our system.

This model presents a number of challenging aspects that can be handled by our approach:

1. block composition;
2. standard ports, interfaces and connectors;
3. state machines as block behaviour;
4. parallel states;
5. join and fork pseudostates;
6. “do activities”;
7. completion transitions;
8. operation calls on ports;
9. signals and signal events;
10. activities as specification of operations;
11. operation calls in activity diagrams;
12. input and output pins in activities;
13. sequence diagrams as scenario specifications;
14. synchronous and asynchronous messages; and
15. parallel fragments in sequence diagrams.

2.4 Final remarks

In this chapter we have given an overview of the two base notations of COMPASS: CML and SysML. We have illustrated CML by a simple example that illustrates the modelling of a clock and the possible design choices available to the user. Finally, we have presented a simple, yet comprehensive, SysML model that includes instances of all diagrams covered by our semantics.

In the next chapter, we provide the foundations for the semantics of SysML models. We describe the assumptions made about the models, the presentation style used in describing the semantics, and the semantics of SysML models as a set of blocks and diagrams.
Chapter 3

Formal semantics of SysML models

In this chapter, we describe the assumptions made in defining the semantics of SysML models. These assumptions take two forms: guidelines and meta-notation. The first acts upon the SysML models themselves, and establish what is required of a model for a CML model to be ascribed to it. The second specifies what is the meta-notation used in defining the translation rules that formalise the semantics of SysML. The specification of the meta-notation is necessary to clarify what is part of the CML model, and what is not.

In Section 3.1, we present the set of guidelines that allows us to provide a formal account of SysML. Section 3.2 describes the metalanguage used for presenting the translation rules that define the semantics of the diagrams. The chapter concludes with Section 3.3 providing an overview of our strategy for the formalisation of SysML models.

3.1 Guidelines

We identify here a subset of the SysML notation and describe guidelines of how it must be used to allow us to define a semantics via a translation to CML. Our subset of SysML includes block definition diagrams, internal block diagrams, state machine diagrams, activity diagrams and sequence diagrams. For block definition and internal block diagrams, the following constructs are considered: blocks, operations, attributes, signals, ports, provided and required interfaces, associations, composition, generalisation.
For state machine diagrams, the supported constructs are: simple and composite states, regions, final states, initial and junction pseudostates, fork and merge pseudostates, transitions, and completion and deferred events. For activity diagrams, we have translation functions for: control nodes, actions without pins, actions with input or output pin, accept and send signal actions, and object flow. The current semantics is restricted to synchronous communications. Finally, for sequence diagrams the supported constructs are: lifelines, messages (synchronous, asynchronous and reply), message arguments, local attributes of the Interaction, CombinedFragments (PAR, STRICT, SEQ, ALT, OPT, BREAK, LOOP, CRITICAL), StateInvariant, InteractionUse and Gates.

Our guidelines of usage aim at supporting the definition of interesting cohesive formal models of SysML models. The guidelines maximise the definedness of a SysML model at both the entity definition level and at the instance definition level. In this respect, they are similar to constraints imposed by most tools to enable automatic code generation from models.

A SysML model typically provides independent (although expected to be consistent) and disconnected views of an application (much like a set of classes in an object-oriented program without a main class that coordinates all other classes). To generate a CML model (or indeed code), we need a connected model with an element (typically a block) that identifies the application as a whole. Our guidelines for maximal definedness at the entity definition level, however, still allow for models that provide independent disconnected views. The guidelines for definedness at the instance level, therefore, require that enough information is available in the SysML model to link models provided by the diagrams. Furthermore, the guidelines include three restrictions that simplify the semantics and make it more amenable to automated reasoning.

**Entity definition guidelines.** They are as follows:

1. The body of each operation must be defined either by the block’s state machine diagram, by a CML statement, or by an activity diagram;
2. In the graph of the composition relation, there must be exactly one connected component of size larger than 1, and this component must be a tree;
3. The blocks in the model must form a connected graph where the edges are either generalisation or composition relations;
4. A composite block must not have attributes, operations, signals, activities or state machines;
5. Associations in the block definition diagram must be matched in the internal block diagram typed connectors;

6. Associations must be used in place of aggregation; and

7. Connectors between ports are not typed.

The first guideline contributes to the maximum definedness requirement by assuring that every operation is defined somewhere in the model. Whilst it is possible to define the model of a block whose operations are not defined, such model would represent only the signature of the operations. Moreover, if an operation has a return value, any possible values can be returned. This potentially restricts the effectiveness of the formal reasoning supported by such models.

The second guideline guarantees that there is a root block of the model from which the semantics of the whole model can be defined, and the third guideline guarantees that all other blocks can be reached from a block in the model. We observe that this is essential to ensure that we have a connected model of the application, but it is not enough, since it is concerned only with block diagrams.

The fourth guideline requires that a composite block is simply the composition of its parts. The reason for this restriction is that it is not clear what it means for a composition of blocks to have an associated behaviour (in the form of a state machine, for instance). We envisage two possible interpretations: (1) the state machine is a specification of the desired behaviour of the composition of the parts, or (2) the state machine specifies a behaviour that is specific to the composite block and must be executed in parallel with the parts of the block.

If the second interpretation is the intention, it is possible to write a model that captures this behaviour and follows our guidelines. We need to create a new part and move all components of the composite block to the new part. In this case, it would be possible to adequately specify the dependencies between the behaviours in the new part with respect to the original parts.

Whilst operations and signals are not allowed in the composite block, ports are supported and operations and signals on the ports are allowed. This restriction forces the user to indicate, through the use of connectors and interfaces, which parts treat which requests received at the ports of the composite block.

The fifth and sixth guidelines enforce our view that whilst composition is a relation between blocks, aggregation and association are relations between instances. We require that associations are used instead of aggregation, and that associations are matched by connectors at the instance level.
Our view is motivated by the fact that composition entails uniqueness in the sense that the same instance cannot be part of two different blocks, and therefore the semantics of the parent block is directly defined in terms of the sub-blocks. Moreover, since the instances of the sub-blocks that compose an instance of the parent block are unique, we do not require any additional information to instantiate them as long as we can guarantee that they are unique.

Aggregation and association on the other hand allow sharing of blocks, and without the uniqueness assumption, we require additional information to determine whether instances are shared and how. Since block definition diagrams cannot refer to particular instances, any aggregation or association present in the block definition diagram must be further specified in the internal block diagram to provide this information. For associations, a connector in the internal block diagram must be typed by an association in the block definition diagram. In the case of aggregation, the same approach is not possible because connectors cannot be typed by aggregations. For this reason, our guidelines do not allow the use of aggregation.

Finally, since ports are not shown in block definition diagrams, connectors where at least one end is a port cannot be typed by associations.

**Instance-level guidelines.** They are as follows:

1. Each composition head must specify the connections between its parts through an internal block diagram.
2. All blocks in a composition whose minimum cardinality is larger than 0 must appear in the internal block diagram of the containing block in numbers compatible with their multiplicities.
3. The cardinalities that appear in an internal block diagram must be constants.
4. Ports may only be connected to other ports.

The internal block diagram specifies exactly which instances are present in a particular realisation of the system, and exactly how they are associated (or not). This is enforced by the first three guidelines above.

The last guideline is necessary because a connector starting in a part (not a port) must correspond to an association of the part, which in turn must appear in the block definition diagram. Since in the COMPASS SysML tool ports cannot be shown in block definition diagrams, the association cannot be between a port and a block, and therefore the connector typed by that association cannot link a port
and a part. In this case, restriction 4 above is simply a consequence of the entity-level restriction 5.

**Action language assumptions.** Since the action language of SysML is not defined, we adopt a subset of CML. SysML operations and actions may be defined using CML statements, that is, they may contain the same constructs used in CML operation definitions. No reactive behaviour is allowed, though. This is because the SysML paradigm is not based on the use of channels for communications, like CML. Instead, attributes and operations of a block define the services provided by an application to its users. SysML operations are, therefore, data operations that explain how the state embedded in blocks and other operations can be used to specify the service.

In the context of state machine diagrams, we extend the subset of CML used as action language to include the following constructs:

- **Block operation call without return value** \( \text{Op}(p_1, \ldots, p_2) \)
- **Block operation call with return value** \( v := \text{Op}(p_1, \ldots, p_2) \)
- **return statement** \( \text{return } v \)

Whilst the syntax of these constructs is similar to the syntax of CML, their semantics is not that of CML. The return statement is only allowed in transition actions triggered by a call event. This restriction is necessary because the return statement is part of the definition of an operation, and as such must be associated with an operation definition. For state machine diagrams, this is the case only for actions triggered by a call event.

**Simplification assumptions.** The only guidelines of usage whose purpose is solely to simplify the semantics are as follows:

1. Each block must either have an associated state machine diagram or no other associated diagram specifying its behaviour;
2. Sequence diagrams specify scenarios of interactions between blocks that the model must or must not allow;
3. Operation are always synchronous. Asynchronous operations should be modelled by signals.

The first guideline establishes that the only valid form of restricting the behaviour of a block is by means of state machines. Essentially, this forbids the use of activity and sequence diagrams for the specification of the overall behaviour of a
block. State machine diagrams are particularly well suited for this task, since they support deferred events (useful to model blocking behaviour) as well as call events (useful to model operations). Moreover, there is a convention [FMS11] for modelling operations that return values (i.e., the return keyword) in a state machine, and our semantic definition of state machine diagrams covers such convention.

The second guideline restricts the possible uses of sequence diagrams to scenario validation purposes. This use is consistent with various case studies and recommendations [FMS11]. Although sequence diagrams have been used for other purposes (e.g., operation definition), it is our view that the emphasis on interaction between blocks given by sequence diagrams makes them more suitable to model such scenarios than, for instance, to model operation definition.

The final guidelines forbids the use of asynchronous operations. The reason for this guideline is that asynchronous operations give rise to some unclarities. It is not clear what is the intended meaning of an asynchronous operations with return values, and the difference between asynchronous operations and signals. If an asynchronous operations is required, we suggest the use of signals as a replacement.

3.2 Semantic rules metalanguage

In this section, we outline the conventions used in the presentation of the translation rules. For instance, Rule [3.1] takes SysML type and outputs CML type. The font differences emphasise the distinction between CML and the metalanguage. For instance, the underlined if statement, is not part of the generated CML models, but the bold face nat may appear in the CML model as the translation of the SysML type Nat.
Rule 3.1 \( t_{\text{types}}(t:\text{Type}): \text{CML\_Type} = \)

\[
\begin{align*}
\text{if } t &= \text{Nat} \text{ then } \text{nat} \\
\text{elseif } t &= \text{Nat1} \text{ then } \text{nat1} \\
\text{elseif } t &= \text{Int} \text{ then } \text{rat} \\
\text{elseif } t &= \text{Real} \text{ then } \text{real} \\
\text{elseif } t &= \text{Bool} \text{ then } \text{bool} \\
\text{elseif } t &= \text{Char} \text{ then } \text{char} \\
\text{elseif } t &= \text{String} \text{ then } \text{seq of char} \\
\text{elseif } t \text{ is a block name then ID} \\
\text{elseif } t &= X[n] \text{ then seq of } t_{\text{types}}(x) \\
\text{else token} \\
\text{end if}
\end{align*}
\]

The definition of the translation rules uses a simple metalanguage; the following conventions are adopted:

- Terms of the CML syntax are presented in the CML style, with black tele-type font, and bold face keywords;
- The translation notation (metalanguage) is presented in a lighter coloured underlined font;
- The type of SysML constructs to which the rule is applicable (that is, the domain of the semantic function that it defines) and the type of the CML constructs generated by the rule (that is, the image of the semantic function that it defines) are specified in the declaration of the function signature (e.g., \( t_{\text{types}}(t:\text{Type}): \text{CML\_Type} \));
- Components of SysML constructs are accessed using a ",." (e.g., \( b_{\text{name}} \) refers to the name of the SysML block denoted by b). Such components are defined in the SysML abstract syntax;
- If and for statements are used with their usual meaning;
- The function \( \text{name} \) takes a SysML object (e.g., state, transition) and produces a unique name that identifies that object. The actual name is not important, as long as it is unique;
- The function \( \text{id} \) is defined as \( \text{mk\_token(\text{name}(s)))} \); it produces a sequence containing a single token formed by the unique name of a SysML object. The unique name is obtained through the application of the function \( \text{name} \);
- We assume that each element of a diagram can be identified uniquely by the field \( \text{index} \), which is accessed using ",." (e.g., \( \text{element\_index} \)). Such field
is important to differentiate two elements inside the same diagram, because we may have the case where elements exist with same name or signature. For the purpose of translations we assume that this field is a natural number, however, implementers may decide to use any type that fits the requirement to identify uniquely a element of the same diagram. In this latter case, is important to change the channel types where this field is used.

- The function \( \text{set2seq}(s) \) takes a set \( s \) and produces a sequence whose elements are those of \( s \);

- Lists whose separators are symbols of the CML syntax are defined by list constructors. For instance, we have a constructor \( [\] \), which takes a set of terms and generates a comma-separated list of these terms. For example, \( [1,2,3] \) is the list \( 1,2,3 \) to be used, for instance, in a set enumeration.

The formalisation of the list constructors is simple and omitted here. We further explain and illustrate its use as needed. All these operators of the metalanguage are identified by boxed symbols.

In the declaration of the translation function signatures, we define the construct of the CML abstract syntax \( \text{[WCC+12]} \) that characterises the range of the function. For example, \( \text{t_block_types}(b: \text{Block}): \text{class declaration} \) takes a SysML block as parameter and defines a CML class.

We call each of the equations that define our translation function a translation rule. This reflects the fact that these definitions can be used as rewrite rules to generate CML models (automatically).

### 3.3 Overview of the semantics of SysML models

The domains of the translation functions are sets of SysML constructs and, as far as possible, we have adhered to the metamodel of SysML presented in the standards for both UML 2.0 and SysML. We do not adhere to the metamodel in situations where the necessary component is not directly available. For instance, we assume a block has a component \( \text{sim} \) which may contain a reference to the state machine associated with the block.

As explained in the Section 3.1, we consider SysML models that contain a number of connected blocks and associated behavioural diagrams defined in accordance with our guidelines of usage. We also assume that the model has one or more root blocks. Under these assumptions, the semantics of a model is defined by
the semantics of all the root blocks. The semantics of a model is given by the following translation function.

**Rule 3.2** \( t_{model}(m: \text{SysML model}): \text{program} = \)

```plaintext
types
   ID = seq of token;
   DL = bool | <defer>;

for each op in m.AllOperations do
   _t_type_operation(op)
end for

for each s in m.AllSignals do
   _t_type_signal(s)
end for

OPS = | for each op in m.AllOperations do
       name(op)_I | name(op)_O
     end for

S = | for each op in m.AllSignals do
     name(op)
     end for

MSG = OPS | S
for each i in m.AllInterfaces do
   _t_interface_types(i)
end for

functions
   DL_or(a: DL, b: DL) c: DL
     post (is_bool(a) and is_bool(b) => c = a or b)
     and (not is_bool(a) or not is_bool(b) => (a = true) or (b = true) => (c = true))
     and (not (a = true) and not (b = true)) => c = <defer>)

) channels
for each c in m.AllConnectors do
   name(c).op: nat*ID*ID*OPS
   name(c).sig: nat*ID*ID*S
end for
for each b in m.AllBlocks do
   if b.isSimple
      then _t_simple_block(b)
   else _t_composite_block_process(b)
   end if
end for
for each sd in m.AllInteractions do
   _t_sequencediagram(sd)
end for
```

It first declares the types \( \text{ID} \) and \( \text{DL} \) as a sequence of elements of type \text{token} and a union type formed by the set of booleans and the quote type \text{<defer>}. The type \( \text{DL} \)
is necessary for encoding state machine, and is the same for all state machine and, therefore, is defined globally. It encodes the possible outcomes of processing an event, the event can be deemed consumed, not consumed or deferred. This type has an associated function \texttt{DL\_or}, which determines whether or not an event is deferred, based on whether two parallel regions consume or defer the event. This type and its associated function are used in Chapter 5.

Next, for each operation in the model, it applies the rule \texttt{t\_type\_operation}, which produces a record type declaration that encode the operation. Similarly to operations, for each signal in the model, a record type is defined by the application of a translation rule. Next all the operation types are gathered in the union type \texttt{OPS}, all the signal types in the type \texttt{S}, and both types are joined to form the type of all messages \texttt{MSG}.

Next, for all interfaces defined in the model, a class is generated by the rule \texttt{t\_interface\_types}, for each connector used in the model, a channel named after it is defined as communicating a natural number, to identifiers and a message. All these channels are internal to the model.

Finally, for each block and sequence diagram in the model, the process that defines it is declared. The result of applying this rule to the example in Section 2.3 is shown below.

\begin{verbatim}
types
  ID = seq of token;
  DL = bool | <defer>;;
  check_I :: ;
  check_O :: b: bool;
  print_I :: s: String;
  print_O :: ;
  fix :: ;
  on :: ;
  off :: ;
  OPS = check_I | check_O | print_I | print_O
  S = fix | on | off
  MSG = OPS | S
  class I\_types = ...
  class IO\_types = ...
  class I\_Monitor\_types = ...

channels
  c\_m1\_p\_ops: nat\*ID\*ID\*OPS
  c\_m1\_p\_sig: nat\*ID\*ID\*S
  c\_m2\_p\_ops: nat\*ID\*ID\*OPS
  c\_m2\_p\_sig: nat\*ID\*ID\*S
  c\_p1\_p3\_ops: nat\*ID\*ID\*OPS
  c\_p1\_p3\_sig: nat\*ID\*ID\*S
  c\_p1\_p2\_ops: nat\*ID\*ID\*OPS

46
\end{verbatim}
c_p1_p2.sig: nat*ID*ID*S

functions
DL_or(a: DL, b: DL) c: DL
post (is_bool(a) and is_bool(b) => c = a or b)
    and (not is_bool(a) or not is_bool(b) =>
        ((a = true) or (b = true) => (c = true))
        and ((not (a = true) and not (b = true)) => c = <defer>)
)

... process Machine = ...
...
process Screen = ...
...
process Monitor = ...
...
process System = ...
...
process sd_pos = ...
...
process sd_neg = ...

The semantics of the whole model is given by the process System, which corresponds to the root block System. Larger definitions and auxiliary declarations covered in the following chapters are elided.

The rule t_types shown in the previous section takes simple SysML types (e.g., Int) and outputs equivalent CML types. A block name used as a type corresponds to the set of instances of that block; it is translated to the type ID, which identifies instances of blocks. A type with multiplicity (i.e., X[n]) is translated into a sequence of the CML types associated with the component type (i.e., X). Finally, any other types are associated with the generic CML type token, which supports only equality comparison. The remaining rules used in t_model are defined and explained in the next chapters.

3.4 Final remarks

In this chapter, we have presented our guidelines of usage of SysML that allows us to formalise the semantics of SysML models, the conventions adopted in the presentation of the translation rules, and the rule that is the root of our translation: t_model. Additionally, we have presented the rule for the translation of types, which is used throughout the next chapters.
The semantics specified by these translation rules and the ones that follow provides a view of the system where the number of instances of blocks as well as the communication structure is fixed. In order to support dynamic creation and destruction of blocks, and reconfiguration of the dynamic structure, we require SysML patterns that support the specification of these behaviours. To the best of our knowledge, such patterns do not yet exist.
Chapter 4

Structural diagrams

In this chapter, we present the semantics of blocks specified by means of block definition diagrams and internal block diagrams. Our translation is based on the underlying SysML model as described by the meta-model of SysML [Obj10a]. In SysML the diagrams are seen as potentially disjoint views of a cohesive model. For instance, the same block may appear in two different diagrams, showing one operation in one diagram, and a different operation on another diagram. The diagrams are two different view of the same block, each emphasising a different operation. The block in the underlying model contains both operations.

The construction of this underlying model is akin to the task of parsing and type checking a CML file. A abstract syntax tree is built from the CML file, elements of the concrete syntax are converted into elements of the abstract syntax, and certain information is inferred from the tree (e.g., that used operations are defined somewhere in the model). In the case of SysML, this task is carried out by the chosen modelling tool, but the resulting abstract syntax tree (underlying model) must respect the abstract syntax (meta-model) specified in the standard of the language.

We exemplify our semantics for blocks through the example in Section 2.3 and a model of the classic producer consumer problem. The block definition diagram for this new model is shown in Figure 4.1. It clearly respects the guidelines described in Section 3.1 and the semantics of the whole diagram is the semantics of the root of the composition tree, that is, the block ProducerConsumer. In particular, in this chapter we use the diagrams shown in figures 4.1, 4.2 and 4.3.

Section 4.1 presents an overview of the semantics of blocks, Sections 4.2 and 4.3 present the translation rules for operations, signals and interfaces. Section 4.4 presents the rules that define the semantics of ports. Finally, Sections 4.5 and 4.6...
define the semantics of simple blocks and composite blocks.

4.1 Overview

The translation of SysML blocks must consider a number of other elements: attributes, signals, operations, interfaces, ports, association, composition, connectors, parts, state machines and activities. The semantics of a block is defined
in terms of the elements that form the block. Figure 4.3 illustrates the possible paths of communication with a block. Operations and signals can be requested, attributes can be read and updated, and events can be received (by a state machine, for instance).

Simple blocks may contain attributes and operations as well as ports. A port may contain a number of interfaces, which are provided or required by the block. An interface on the other hand may contain a number of signals and operations. Finally, simple blocks may have a state machine diagram and a number of activity diagrams, some of which may be associated with operations.

The semantics of a simple block is given by a process that is constructed using the processes that model its components. These are put together through parallel composition, and the alphabets (channel sets) of the processes determine how they communicate. The translation rule for a simple block, declares all the necessary components: types, channels and processes.

The translation of a simple block produces three processes. The first is a simple model of the signature of the block (i.e., its operations and attributes) without taking into consideration generalisation. The second is the so called bare model of the block that combines the encoding of the signature of the process and the bare model of any parent block. Generalisation is modelled by interleaving. The third process combines the bare model of the block with its ports and state machines, if any exist.

Composite blocks may contain other blocks (by composition) and ports. As for
simple block, the ports may contain interfaces, which may have operations and signals. Additionally, the different blocks that form the composite block may be linked by connectors directly or through ports.

Similarly to the case of simple block, the semantics of a composite block is a process that composes in parallel the elements that form it, that is, its parts and ports. The connectors are modelled by channels (see Rule [3.2]), which are used to rename the channels of the parts and link them. In this case, the translation rule also declares the necessary components (i.e., types, channels and processes), except for the processes the model the parts, as these have already been declared by the Rule [3.2].

The process that models the core of a block (its operations and signals) is ready to receive signals and operation calls, and to send operation responses. Every time an operation call occurs, the block registers all the possible responses and delegates the decision of which is the appropriate response to its internal behaviour (or its parts). The possible responses are recorded in a bag (i.e., a set with repetition), whose type is declared for each block as a bag of the block’s operations.

Ports are modelled by processes similar to those that model simple blocks, but since ports only contain interfaces, they are not composed of any other processes.
Interfaces are simply a form of collecting operations and signals, and as such are modelled as types in a CML class that provides the appropriate scope. A port simply relays requests and responses to the appropriate element of the model (i.e., the block’s state machine, a part of a composite block or another port). The nature of the interfaces (provided or required) determine which communications can occur between the port, the block (and its parts) that contains it, and the external environment.

State machines (specified by state machine diagrams) and activities (specified by activity diagrams) similarly yield processes and are the subject of chapters 5 and 7.

### 4.2 Operations and signals

Each operation in the model produces two record types that encode the input and output values of the operation, and each signal produces a single record type. These types are declared by the following translation rules 4.1 and 4.2.

**Rule 4.1** `t_type_signal(s: Signal): record type declaration =`

```
public
name(s) ::
for p: s.params do
  name(p): t_types(p.types)
end for
;`n
```

The record type associated with a signal is named after the signal by the function `name`, which is assumed to provide unique names to the different elements of the model. For each parameter of the signal, a record component of the appropriate type is declared. The result of applying this rule to the signals of model of the monitored machine example is as follows.

```
public fix :: ;
public on :: ;
public off :: ;
```

These record types are all empty since none of the signals have parameters.
Rule 4.2 \( t\text{\_type\_operation}(\text{op}: \text{Operation}) \): seq of record type declaration =

\[
\begin{align*}
\text{public } \text{name(}\text{op}\text{)}\_I :: \\
& \text{for } p: \text{op}\text{.input\_params do} \\
& \quad \text{name(}p\text{)} : t\_\text{types}(p\text{.type}) \\
& \text{end for} \\
; \\
\text{public } \text{name(}\text{op}\text{)}\_O :: \\
& \text{for } p: \text{op}\text{.output\_params do} \\
& \quad \text{name(}p\text{)} : t\_\text{types}(p\text{.type}) \\
& \text{end for} \\
& \text{if } \text{op}\text{.return }!=\text{nil} \text{ then} \\
& \quad _\text{ret}: t\_\text{types}(\text{op}\text{.return\_type}) \\
& \text{end if} \\
; \\
\end{align*}
\]

An operation in SysML can have three types of parameters: \textbf{in}, \textbf{out} and \textbf{inout}. The first and second are the input and output parameters. The third are parameters that are used for input and output. An operation yields two record types, both named after the operation (using \texttt{name}), but the first suffixed by \_I indicating that this type encodes the input parameters of the operation, and the second suffixed by \_O indicating that this type encodes the output parameters. The definition of the records is similar to that of the the records that encode signals, but in the first record only the input parameters are considered; these include parameters classified as \textbf{in} and \textbf{inout}, that is, input and input-output parameters. The second record includes all output parameters (those classified as \textbf{inout} and \textbf{out}) as well as the return value, if the operation has one. The return parameter is always named \_ret. The result of applying this rule to the operation \texttt{check} in the model of the monitored machine is as follows.

\[
\begin{align*}
\text{public } \text{check}\_I :: ; \\
\text{public } \text{check}\_O :: b : \texttt{bool}; \\
\end{align*}
\]

The operation \texttt{check} does not have a return parameter.

All operation and signal types are declared globally. Since the hierarchical structure of the model is lost, name clashes can occur. We assume that the naming function \texttt{name} produces unique names; this is possible for instance by encoding hierarchical information in the name.
4.3 Interfaces

As previously mentioned, an interface generates a CML class. This class declares the types \( I \), \( O \), \( OPS \) and \( S \) of all (input and output) operations and signals contained in the interface. The translation rule for interfaces is shown below.

**Rule 4.3**

\[
\text{t\_interface\_types}(i: \text{Interface}) = \text{class} \begin{array}{c}
\text{name}(i) \_\text{types} = \\
\text{begin} \\
\text{types} \\
\text{if } \text{i\_Signals\_size}(i) > 0 \text{ then} \\
\text{t\_interface\_signal\_type}(i) \\
\text{else} \\
\text{public} S = \text{token inv} x == \text{false}; \\
\text{end if} \\
\text{if } \text{i\_Operations\_size}(i) > 0 \text{ then} \\
\text{t\_interface\_input\_type}(i) \text{ \ t\_interface\_output\_type}(i) \\
\text{else} \\
\text{public} I = \text{token inv} x == \text{false}; \\
\text{public} O = \text{token inv} x == \text{false}; \\
\text{end if} \\
\text{public} \text{OPS} = I | O; \\
\text{end}
\end{array}
\]

It declares a CML class named after the interface. The class contains four type declarations: three for operations and one for signals. The input operation types are gathered in a type \( I \), the output operations are gathered in a type \( O \) and the signal types are gathered in a type \( S \). If there are no operations or signals, the appropriate types (\( I \) and \( O \), or \( S \)) are declared as empty types (\( \text{token inv} x == \text{false} \)). Finally, the input and output operation types are grouped in the \( OPS \) type.

The interface \( I\_\text{Monitor} \) in Figure 4.3 contains a single operation \( \text{check} \) with an out parameter \( b \) of type bool, and the interface \( I \) contains three signals (\( \text{fix}, \text{on} \) and \( \text{off} \)) and no operations. The types associated with them are shown below.

\[
\text{class} \begin{array}{c}
I\_\text{Monitor}\_\text{types} = \begin{array}{c}
\text{begin} \\
\text{types} \\
\text{public} S = \text{token inv} x == \text{false}; \\
\text{public} I = \text{check}_I; \\
\text{public} O = \text{check}_O; \\
\text{public} \text{OPS} = I | O; \\
\text{end}
\end{array}
\end{array}
\]

55
class I_types = begin
    types
        public S = fix | on | off;
        public I = token inv x == false;
        public O = token inv x == false;
        public OPS = I | O;
    end

The input, output and signal types are declared by Rules 4.4, 4.5 and 4.6. The following rule defines the function that declares the type \( S \) of all the signals in the interface.

**Rule 4.4 t_interface_signal_type(i: Interface): seq of class paragraph =**

\[
\text{public } S = \prod \text{for each } s: i.\text{Signals do }
    \text{name}(s) \text{ end for ;}
\]

The rule below declares the input type of all the operations in the interface.

**Rule 4.5 t_interface_input_type(i: Interface): seq of class paragraph =**

\[
\text{public } I = \prod \text{for each op: i.Operations do }
    \text{name(op)}_I \text{ end for ;}
\]

The rule below similarly declares an output type for all the operations.

**Rule 4.6 t_interface_output_type(i: Interface): seq of class paragraph =**

\[
\text{public } O = \prod \text{for each op: i.Operations do }
    \text{name(op)}_O \text{ end for ;}
\]

The assumption about the uniqueness of names of operations and signals of blocks applies to those of interfaces.
4.4 Standard ports

As already said, a standard port is a mechanism for communication between blocks without a specific reference to a block instance. We model a standard port similarly to a simple block, except that it does not treat the received request: it simply relays it to the appropriate block.

For the purpose of our semantics, we distinguish two sides of a port: external and internal. If the port belongs to a simple block, then the internal side is connected implicitly to the block itself. If the port belongs to a composite block, it must be explicitly connected to a port of one of the parts of the block.

The set of provided and required interfaces connected to a port determine the possible patterns of communication of a port. In the most general case, a port has a set of provided interfaces \( PI \) and a set of required interfaces \( RI \). For instance, for the port \( p_1 \) in Figure 4.3, we have the following sets:

\[
RI = \{I\} \\
PI = \{IO\}
\]

These sets of interfaces are determined with respect to the external side of the port and are implicitly mirrored on the internal side. A provided interface is mirrored in a required internal interface on the same port, and a required interface is mirrored in a provided internal interface on the same port. For instance, the provided interface \( I \) on port \( p_3 \) is implicitly mirrored as a required interface on port \( p_3 \).

In our CML model, a port gives rise to four channels of communication (two for operations and two for signals) associated with its sides: internal and external. Signals are asynchronous communications, whereas operations are considered synchronous in our semantics. The patterns of communication on these channels depend on the nature of the requests. Operations in a provided interface can only be requested on the external operation channel, and operations in a required interface can only be requested on the internal channel.

The types associated with a port are those of its interfaces. They are grouped according to the type of interface (provided or required), and whether the type refers to a signal or operation. The following rule declares the types associated with a port and is used in the definition of a port as specified by the Rule 4.7.
The provided operation and signal types \((P_{\text{OPS}}\text{ and } P_{\text{S}})\) are built from the operations and signals of the provided interface (e.g., \(I_{\text{Monitor}}'\text{OPS}\)). Similarly, the required operation and signal types \((R_{\text{OPS}}\text{ and } R_{\text{S}})\) are built from the required interfaces. The set of all operations (both provided and required) is named \(\text{OPS}\) and the set of all signals is named \(\text{S}\).

The four channels used by a port are declared by the following rule.

**Rule 4.8** \(t\_\text{port_channels}(p: \text{Port})\): channel declaration =

```plaintext
channels
    name(p).int_sig: nat*ID*ID*S
    name(p).ext_sig: nat*ID*ID*S
    name(p).int_op: nat*ID*ID*OPS
    name(p).ext_op: nat*ID*ID*OPS
```

The first two channels, \(\text{int\_sig}\) and \(\text{ext\_sig}\), communicate signals in and out of the port; they communicate a natural number identifying a particular request (necessary to distinguish otherwise identical requests made by the same block),
the identifiers of the source and target of the send signal action, and the encoding of the signal as an instance of the appropriate record type. Similarly for operations, the two remaining channels \( \text{int}_\text{op} \) and \( \text{ext}_\text{op} \) communicate a natural number identifying the instance of the operation call, the identifiers of the source and target of the call, and the operation call itself, in the form of an input or output type. Each operation call involves two communications on the channel, the first with an input value and the second with the output value.

The process that models a particular port is declared by the following rule; it is used in the definition of both simple and composite blocks.

**Rule 4.9** \( \text{t}_\text{port}(p: \text{Port}): \text{seq of paragraph}= \)

\[
\begin{align*}
\text{t}_\text{port}_\text{types}(p) \\
\text{t}_\text{port}_\text{channels}(p)
\end{align*}
\]

\[
\text{process port}_\text{name}(p) = \text{id: ID @ begin}
\]

\[
\begin{align*} &@ \mu \text{X @ (} \\
&\quad \text{name}(p).\text{ext}_\text{sig}?i?o!id?x:(\text{is}_\text{name}(p)\text{)}\text{types}'P_S(x)) \rightarrow \\
&\quad \text{name}(p).\text{int}_\text{sig}.i?y.x \rightarrow \text{Skip} \\
&\quad () \\
&\quad \text{name}(p).\text{int}_\text{sig}?i?o!id?x:(\text{is}_\text{name}(p)\text{)}\text{types}'R_S(x)) \rightarrow \\
&\quad \text{name}(p).\text{ext}_\text{sig}.i?y.x \rightarrow \text{Skip} \\
&\quad () \\
&\quad \text{name}(p).\text{ext}_\text{op}?i?o!id?x:(\text{is}_\text{name}(p)\text{)}\text{types}'P_I(x)) \rightarrow \\
&\quad \text{name}(p).\text{int}_\text{op}.i?y.x \rightarrow \text{Skip} \\
&\quad () \\
&\quad \text{name}(p).\text{int}_\text{op}?i?o!id?x:(\text{is}_\text{name}(p)\text{)}\text{types}'P_O(x)) \rightarrow \\
&\quad \text{name}(p).\text{ext}_\text{op}.i!y.x \rightarrow \text{Skip} \\
&\quad () \\
&\quad \text{name}(p).\text{int}_\text{op}?i?o!id?x:(\text{is}_\text{name}(p)\text{)}\text{types}'R_I(x)) \rightarrow \\
&\quad \text{name}(p).\text{ext}_\text{op}.i!y.x \rightarrow \text{Skip} \\
&\quad () \\
&\quad \text{name}(p).\text{ext}_\text{op}?i?o!id?x:(\text{is}_\text{name}(p)\text{)}\text{types}'R_O(x)) \rightarrow \\
&\quad \text{name}(p).\text{int}_\text{op}.i!y.x \rightarrow \text{Skip} \\
&\quad ()
\end{align*}
\]

\[
\}; X \\
\text{end}
\]

The main action of the process, which is named after the port, defines that it receives signal events of the provided interfaces on the external channel, and forwards them through the internal channel. Conversely, signal events of the required
interfaces are received only on the internal channel and forwarded on the external channel.

The relaying of an operation call is slightly more complicated because the relaying of inputs and outputs is carried separately. Input records of operation calls of the provided interface are only received through the external channel and relayed through the internal channel. Output records of operation calls of the provided interface are received through the internal channel and forwarded through the external channel. Input and output records of operation calls of the required interface are received and sent in a complementary way. Inputs are received in the internal channel (and forwarded on the external channel), and output are received in the external channel (and forwarded on the internal channel).

Figure 4.5: Allowed communication patterns for a port.

Figure 4.5 shows the patterns of communication between the external and internal environments through a port. The port has a provided interface containing the operation $\text{Op2}$ and the signal $\text{S2}$, and a required interface containing the operation $\text{Op1}$ and the signal $\text{S1}$. The input of $\text{Op2}$ can only go inwards, and the output can only go outwards. The input of $\text{Op1}$ can only go outwards, and the output can only go inwards. The signals $\text{S1}$ and $\text{S2}$ can only go, respectively, outwards and inwards.
4.5 Simple blocks

The next set of rules applies to simple blocks, that is, blocks that are not composed of other blocks. Whilst simple blocks cannot be the source of a composition relation, they can be part of another block (target of a composition relation) and can be associated with other blocks through generalisation and association. These aspects are dealt with by the translation rule for composite blocks.

The root translation rule is \texttt{t\_simple\_block}. It takes a simple SysML block and produces a number of definitions: types, channels and processes. It produces these definitions by calling other translation rules, namely, \texttt{t\_block\_types}, \texttt{t\_block\_channels}, \texttt{t\_port} and \texttt{t\_port\_process}, on the block. Each of these translation rules is further defined below.

Rule 4.10 \texttt{t\_simple\_block(b: Block): seq of program paragraph =}

\begin{verbatim}
t_block_types(b)
t_block_channels(b)
for each p in set b.Ports do
\quad t_port(p)
end for
\quad t_block_process(b)
\end{verbatim}

The result of applying this rule to the block \texttt{Buffer} is shown after the presentation of each of the rules \texttt{t\_block\_}. 

4.5.1 Simple blocks: types

Similarly to the rule for interface, \texttt{t\_block\_types} declares a class whose name is formed by suffixing the name of the block with \texttt{\_types}. The class declares a type \texttt{I} as the union type of the input types of the operations in the block, the type \texttt{O} as the union type of the output types of the operations in the block and the type \texttt{S} as the union type of the types associated with the signals in the block. Finally, the class declares a type \texttt{OPS} as the union type of all the types associated with the operations in the block, and specifies a bag type and its operation types.
Rule 4.11  \( t\_\text{block}\_\text{types}(b:\text{Block}): \text{class\ declaration =} \)

```plaintext
class name(b) _types =
if b.parent <> NULL then
extends name(b.parent) _types
end if
begin
  types
  if b.Signals.size() > 0 then
    t_block_signal_type(b)
  else
    if b.parent = NULL then
      public S = token inv x == false;
    end if
  end if
  if b.Operations.size() > 0 then
    t_block_input_type(b)
    t_block_output_type(b)
  else
    if b.parent = NULL then
      public I = token inv x == false;
      public O = token inv x == false;
    end if
  end if
  public OPS = I | O;
  declare_bag()
end
```

The rules \( t\_\text{block}\_\text{signal}\_\text{type} \), \( t\_\text{block}\_\text{input}\_\text{type} \) and \( t\_\text{block}\_\text{output}\_\text{type} \) are similar to the rules for interfaces, except that they take into account the generalisation relation. The full specification of these rules can be found in Appendix A.

In case the block does not contain operations, the types \( I \) and \( O \) are declared as empty types (token types with a false invariant). This assures that it is always possible to recover the types \( S \), \( I \) and \( O \) from a parent, even when they are empty.

The class generated by the rule \( t\_\text{block}\_\text{types} \) declares a type \( \text{Bag} \) that contains elements of the type \( \text{OPS} \). It is declared by the following rule.
Rule 4.12 declare_bag(): seq of class paragraph =

```plaintext
public Bag = map OPS to nat
values
    public empty_bag = { |-> }
functions
    public in_bag: OPS * Bag -> bool
    in_bag(o,b) = (o not in set dom b) or
        (o in set dom b and b(o) = 0);
    public bunion: Bag * Bag -> Bag
    bunion(m1,m2) ==
        { x |-> y | x in set (dom m1 inter dom m2), y: nat @
            y = m1(x)+m2(x)}
    munion
        { x |-> y | x in set dom m1, y: nat @
            x not in set dom m2 and y = m1(x)}
    munion
        { x |-> y | x in set dom m2, y: nat @
            x not in set dom m1 and y = m2(x)];
    public bdiff: Bag * Bag -> Bag
    bdiff(m1,m2) ==
        { x |-> y | x in set (dom m1 inter dom m2), y: nat @
            y = if m1(x)-m2(x) > 0 then m1(x)-m2(x) else 0 }
    munion
        { x |-> y | x in set dom m1, y: nat @
            x not in set dom m2 and y = m1(x)};
```

The type `Bag` models a bag (i.e., an unordered collection of objects where repetition is allowed) of operation records. Additionally, three functions are declared: `in_bag`, `bunion` and `bdiff`. These functions are the bag equivalent of set membership, union and difference respectively.

The application of `__block_types` to the block `Buffer` is as follows.

```plaintext
class Buffer_types = begin
    types
        public I = rem_I | add_I;
        public O = rem_O | add_O;
        public S = token inv x == false
        public OPS = I | O;
        public Bag = map OPS to nat
    values
        public empty_bag = { |-> }
    functions
        public in_bag: MSG * Bag -> bool
        in_bag(o,b) = (o not in set dom b) or
            (o in set dom b and b(o) = 0);
```

63
public bunion: Bag * Bag -> Bag
bunion(m1,m2) ==
{x |-> y | x in set (dom m1 inter dom m2), y: nat @
y = m1(x)+m2(x)}
munion (x |-> y | x in set dom m1, y: nat @
x not in set dom m2 and y = m1(x})
munion (x |-> y | x in set dom m2, y: nat @
x not in set dom m1 and y = m2(x));

public bdiff: Bag * Bag -> Bag
bdiff(m1,m2) ==
{x |-> y | x in set (dom m1 inter dom m2), y: nat @
y = if m1(x)-m2(x) > 0 then m1(x)-m2(x) else 0}
munion (x |-> y | x in set dom m1, y: nat @
x not in set dom m2 and y = m1(x));

end

4.5.2 Simple blocks: channels

The channels associated with a block are the op channel, the sig channel, the addevent channel, and getter and setter channels for each attribute of the block.

The op and sig channels are used for communication between the environment and the block (passing the identifier of the particular request, the source and target of the request as well as operation or signal requests), while the addevent channel is used to communicate received requests to a state machine associated with the block or activity diagrams associated with operations. The getter and setter channels reflect the names of the block and its attribute and are used internally to provide concurrent access to the attributes by the processes that model the block operations.

Rule 4.13 t_block_channels(b: Block): channel declaration =

<table>
<thead>
<tr>
<th>channels</th>
</tr>
</thead>
<tbody>
<tr>
<td>name(b).op: nat<em>ID</em>ID*OPS</td>
</tr>
<tr>
<td>name(b).sig: nat<em>ID</em>ID*S</td>
</tr>
<tr>
<td>name(b).addevent: nat<em>ID</em>ID*MSG</td>
</tr>
<tr>
<td>for a: b.Attributes do</td>
</tr>
<tr>
<td>name(b).get_name(a): ID<em>ID</em>t_types(a.type)</td>
</tr>
<tr>
<td>name(b).set_name(a): ID<em>ID</em>t_types(a.type)</td>
</tr>
<tr>
<td>end for</td>
</tr>
</tbody>
</table>
The application of `t_block_channels` to the block `Buffer` gives the following declaration.

channels

- `Buffer.op: nat*ID*ID*OPS`
- `Buffer.sig: nat*ID*ID*S`
- `Buffer.addevent: nat*ID*ID*MSG`
- `Buffer.get_b: ID*ID*seq of token`
- `Buffer.set_b: ID*ID*seq of token`

### 4.5.3 Simple blocks: processes

The translation rule for blocks uses five other translation rules: `t_block_simple_process` to produce the simple model of the block, `t_block_bare_process` to produce the bare model of the block, `t_activity_diagrams` and `t_activities_chanset` to generate the model of activity diagrams and the set of external events associated with the block’s activities (see Chapter [7]), and `t_statemachine` to generate the model of a state machine (see Chapter [5]).

If a state machine is present, it is combined in parallel with the bare model of the block to form the process named after the block. If there are any activity diagrams, the process that models them is composed in parallel with the block (and state machine) model. The synchronisation set is determined by the allowed external communications of the block’s activities. If there are any ports in the block, the models of the ports are composed in interleaving, and the interleaved models are composed with the block (as well as activities and state machine) model in parallel. The interaction patterns between the different processes are determined by the synchronisation sets in the parallel compositions.

The parallelism between a bare model and the model of a state machine involves the channels `addevent` and the events associated with the channel `op` where the second parameter is the identifier of the block, and the fourth parameter is a value of an output operation type. The synchronisation set of the parallelism between the block model (bare model plus state machine) and the ports contains all the events associated with the channel `op` where the second and third parameters are the identifier of the block and the identifier of one of the ports (in any order). Since the model of a port does not use the same channel `op` as the block, its internal channel (`int_op`) is renamed to match the blocks `op` channel.
Rule 4.14 \( t\_block\_process(b: \text{Block}): \text{seq of process declaration} = \)

\[
\begin{align*}
t\_block\_simple\_process(b) \\
t\_block\_bare\_process(b) \\
\text{if } b\text{.activities.size()} > 0 \text{ then} \\
\quad t\_activity\_diagrams(set2seq(b\text{.activities}), b) \\
\text{end if} \\
\text{if } b\text{.stm} == \text{NULL} \text{ then} \\
\quad \text{process } \text{name(b)} = \text{id}: \text{ID} @ (\{( \text{bare\_name(b)} (\text{id}) \}) \\
\quad \text{else} \\
\quad \text{t\_statemachine(b.stm)} \\
\quad \text{process } \text{name(b)} = \text{id}: \text{ID} @ (\{( \text{bare\_name(b)} (\text{id}) \}) \\
\quad \quad \text{stm\_name(b.stm)(\text{id}^{<\text{stm}})} \\
\text{end if} \\
\text{if } b\text{.activities.size()} > 0 \text{ then} \\
\quad \text{t\_activities\_chanset(b\text{.activities}, b)} \\
\quad \text{name(b)}\_ads(\text{id}) \\
\text{else} \\
\text{end if} \\
\text{if } b\text{.Ports.size()} > 0 \text{ then} \\
\quad \{( \text{name(b)}\_op\_n\_id\_p, \text{name(b)}\_op\_n\_p\_id \} \\
\quad \quad \text{p} \in \text{set} \{ \{ \text{id}^{\text{id}(x)} | x \in \text{set b.Ports} \}, n: \text{nat} \} \\
\quad \{( \text{set} \{ \text{id}^{\text{id}(p)} | p \in \text{set b.Ports} \}) \\
\quad \quad \{( \text{name(b)}\_n\_op\_id\_p, \text{name(b)}\_n\_op\_p\_id \} \\
\quad \quad \quad \text{p} \in \text{set} \{ \{ \text{id}^{\text{id}(x)} | x \in \text{set b.Ports} \}, n: \text{nat} \} \\
\text{else} \\
\text{end if}
\end{align*}
\]

The rule for simple processes shown below uses three other translation rules, one to produce the state of the simple process, the second to produce the action that model the access to the state variables, and the third to produce the action that
receives requests to the operations of the block.

**Rule 4.15**

\[
\text{t\_block\_simple\_process}(b: \text{Block}): \text{process declaration} =
\]

\[
\text{process simple\_name}(b) = id: \text{ID} @ \text{begin}
\text{t\_block\_state}(b)
\text{actions}
\text{t\_block\_state\_action}(b)
\text{t\_block\_requests\_action}(b)
@ \text{name}(b)\_state
\{{\name(a) | a \in \text{set b.Attributes}}\} || \{\text{enabled}\}
\text{name}(b)\_requests
\text{end}
\]

The state access and the operation control actions are combined in interleaving with a partition of the state that gives control of the components associated with the block’s attributes to the access action, and the component enabled to the requests action.

The rule for bare processes checks whether the block has a parent or not, and if it does, composes the bare process of the parent with the simple process of the block in interleaving.

**Rule 4.16**

\[
\text{t\_block\_bare\_process}(b: \text{Block}): \text{process declaration} =
\]

\[
\text{if } b.\text{parent} == \text{NULL} \text{ then}
\text{process bare\_name}(b) = id: \text{ID} @ \text{simple\_name}(b)(id)
\text{else}
\text{process bare\_name}(b) = id: \text{ID} @
\text{simple\_name}(b)(id)
||
\text{bare\_name}(b.\text{parent})(id) \{ [\text{name}(b.\text{parent}) <- \text{name}(b)] \}
\text{end if}
\]

If the block does not have a parent, its bare process is exactly the same as its simple process.

The application of \_block\_process to the block Buffer results in the following specification.
process simple_Buffer = id: ID @
   ... 
   actions
   ...
   @ Buffer_state [(b)||{enabled}] Buffer_requests
end

process bare_Buffer = id: ID @ simple_Buffer(id)
...
process Buffer = id: ID @ ( 
   bare_Buffer(id)
   [||{Buffer_addevent, Buffer.op.n.id.x.y | 
   x: ID, y: Buffer_types'O, n: nat}||]
   stm_Buffer(id^<stm>)

The state of the simple process is declared by the rule t_block_state. For each attribute of the block, a state component is declared with the appropriate name and type. Furthermore, a bag of output types (associated with the operations of the block) is declared to record which outputs are enabled. This component is initialised with the empty bag.

Rule 4.17 t_block_state(b: Block): state declaration =

state
   for a: b.Attributes do
      name(a): t_types(a.type);
   end for
   enabled: name(b)_types'Bag := name(b)_types'empty_bag;

This rule is applied to the block Buffer to declare the state components of the process simple_Buffer as follows.

process simple_Buffer = id: ID @
   state
      b: seq of Item;
      enabled: Buffer_types’Bag := Buffer_types’empty_bag;
   actions
   ...
   @ Buffer_state [(b)||{enabled}] Buffer_requests
end

The rule t_block_state_action declares a recursive (mu X) action that offers to read and write the blocks attributes using the channels get_ and set_.
**Rule 4.18** \( t\_\text{block\_state\_action}(b:\ \text{Block}) = \)

\[
\text{name}(b)\_\text{state} = \mu X @ ( \\
\text{for} \ a:\ b\text{.Attributes} \ \text{do} \\
\quad \text{name}(b)\_\text{get\_name}(a)?o!id!\text{name}(a) -> X \\
\quad \left[\right] \\
\quad \text{name}(b)\_\text{set\_name}(a)?o!id?x -> \text{name}(a) := x; X \\
\text{end for} \\
) 
\]

This rule contributes to the definitions of the actions of the process \text{simple\_Buffer}.

\text{process} \text{simple\_Buffer} = \text{id}: \text{ID} @ \text{begin} \\
\text{state} \\
\quad b: \text{seq of Item}; \\
\quad \text{enabled: bag of Buffer\_types}'O := [||]; \\
\text{actions} \\
\quad \text{Buffer\_state} = \mu X @ ( \\
\quad \text{Buffer}\_\text{get\_b}(b)?o!id!b -> X \\
\quad \left[\right] \\
\quad \text{Buffer}\_\text{get\_b}(b)?o!id?x -> b := x; X \\
\text{end for} \\
\text{...} \\
\text{@ Buffer\_state }[\left(b\right)||||] \text{ Buffer\_requests} \\
\text{end} \\
\]

Similarly to the previous rule, \( t\_\text{block\_requests\_action} \) declares a recursive action that waits for a request to start and end an operation on the \text{op} channel. If it receives a request to start an operation, it forwards the request through the \text{addevent} channel, adds all possible answers to the bag \text{enabled} and recurses. If the request is to end an operation that is in the bag \text{enabled}, it will remove the items that were added to the bag when the operation was first requested. Finally, if the request is for a signal, it simply sends the request through the \text{addevent} channel.
In our example, the only process that intercepts communication on the channel `addevent` is the state machine, but in a model with activities, they may intercept the communications on `addevent` as well. In this case, the alphabets of the process would control which process effectively intercepts which `addevent` communications. The result of applying this rule to our buffer example completes the definition of the process `simple_Buffer` as follows.

```plaintext
process simple_Buffer = id: ID @
begin
state
b: seq of Item;
enabled: Buffer_types'Bag := Buffer_types'empty_bag;

actions

Buffer_state = mu X @ {
    Buffer.get_b?o!id!b -> X
}[]
Buffer.get_b?o!id?x -> b := x; X
}

Buffer_requests = mu X @ {
    Buffer.op?n?o!id?x:(is_Buffer_types'I(x)) -> (}
This process declares a state formed by the single attribute $b$ of the block Buffer and the set of enabled output enabled. The main action of the process is the interleaving of two actions: Buffer_state and Buffer_requests. The former controls the state and allows communications that read and write to the components $b$, and the latter controls the receipt of operation and signal requests. In the interleaving, the state is partitioned between the two actions: the component corresponding to the block’s attribute is assigned to the first action, and the component enabled to the second action.

The action that controls the operation and signal requests, receives a value of an operation input type through the channel Buffer.ops or a value of a signal type through the channel Buffer.sig. In the first case, it identifies which operation has been called, sends the call through the channel Buffer.addevent, and adds all possible answers to the bag enabled, and recurses. In the case of a signal value, the action sends the signal through the channel Buffer.addevent, and recurses.

The rule t_statemachine is defined in the following chapter and its application to our example is illustrated throughout the chapter.
4.6 Composite blocks

Composite blocks are those that contain part properties, that is, those blocks that are formed of other blocks through the composition relation. For example, the model of the producer consumer problem on Figure [5.2] has a composite block, namely `ProducerConsumer`, and the model for the monitored machine has a composite block called `System`.

Our guidelines require that a composite block does not have behaviours or attributes, therefore we must only treat ports in the composite system boundary. Notice that whilst we do not treat composite blocks with behaviours, it is possible to add a new part to the composite block, move all behaviours to the new block, and connect the part to any ports or other parts as necessary.

A composite state is modelled by the alphabetised parallel composition of the processes that model its parts, including the outer ports (i.e., the ports of the composite state), where the alphabets are defined by the connectors originating from the part and its ports. The process that models a part is that derived from the block that types the part. The external channels of the ports in the parts of the composite state must be renamed using the names of the connectors associated with the port.

**Rule 4.20**

```
rule t_composite_block_process(b: Block): process declaration =

for p in set b.Ports
    t_port(p)
end for

process name(b) = id: ID @
    define_alphabetised_parallel(
        [ ] for p in set b.Parts
            (name(p.Type) id^id(p.PartName)) [ [t_rename_ext_ports(p)] ]
        t_chanset_part(p, id)
    end for
    for p in set b.Ports
        .(name(p) id^id(p)) [ [t_rename_int_ports(p)] ]
        t_chanset_part(p, id)
    end for
})
```

The parallel action is defined by a rule, `define_alphabetised_parallel`, that takes a sequence of pairs formed by an action name and a channel set name, and recursively
constructs the alphabetised parallelism of all the actions.

**Rule 4.21**  
\[
\text{define\_alphabetised\_parallel}(\text{pairs}: \text{set of (Name*Chanset)}): \text{action} = \\
\begin{cases}
\text{Skip} & \text{if len pairs} = 0 \\
\text{pairs(1).1} & \text{if len pairs} = 1 \\
\text{pairs(1).1} \text{\{define\_union\_chansets(tl pairs)\}} \text{\{define\_alphabetised\_parallel(tl pairs)\}} & \text{else}
\end{cases}
\]

Since the alphabetised parallelism is a binary operator, we need to compose a number of alphabetised parallel operators hierarchically. To do so, we must calculate the channel set associated with an alphabetised parallelism. This channel set is the union of the channel sets in the parallel operator. To calculate the channel set associated with the alphabetised parallelism formed by a sequence of action names and channel set names, we define a recursive rule that produces the union of all the channel set names in the sequence of pairs of action and channel set names.

**Rule 4.22**  
\[
\text{define\_union\_chansets}(\text{pairs}: \text{seq (Name*Chanset)}): \text{binary expression} = \\
\begin{cases}
\{\|\} & \text{if len pairs} = 0 \\
\text{pairs(1).2} & \text{if len pairs} = 1 \\
\text{pairs(1).2} \text{ union define\_union\_chansets(tl pairs)} & \text{else}
\end{cases}
\]

The translation rule that generates the renaming pairs for a part’s ports is shown below: for each port in the part, it obtains the set of connectors associated with the port and renames the port’s external channel with channels named after the connectors.
Rule 4.23 \( t\text{\_rename\_ext\_ports}(p: \text{Part}): \text{seq of renaming pair} = \)

\[
\begin{align*}
&\text{for each } x \text{ in } p.\text{Ports do} \\
&\quad\text{for each } c \text{ in } x.\text{Connectors do} \\
&\quad\quad\text{name}(p.\text{Type}).\text{ext\_sig} \leftarrow \text{name}(c).\text{sig}, \\
&\quad\quad\text{name}(p.\text{Type}).\text{ext\_op} \leftarrow \text{name}(c).\text{op} \\
&\quad\text{end for} \\
&\text{end for}
\end{align*}
\]

The pairs generated by this rule when applied to the part \text{screen} of our example is shown below.

\[
p2.\text{ext\_sig} \leftarrow c\_p1\_p2.\text{sig}, \; p2.\text{ext\_op} \leftarrow c\_p1\_p2.\text{op}
\]

Since a port may have multiple connectors, the external channel may be renamed multiple times. This means that a communication on the external channel is substituted by a choice of communication over the channels renaming it.

The rule for renaming the internal channels of a port is similar, but simpler. It only considers its own channels, as a port does not contain other ports.

Rule 4.24 \( t\text{\_rename\_int\_ports}(p: \text{Port}): \text{seq of renaming pair} = \)

\[
\begin{align*}
&\text{for each } c \text{ in } x.\text{Connectors do} \\
&\quad\text{name}(p).\text{int\_sig} \leftarrow \text{name}(c).\text{sig}, \\
&\quad\text{name}(p).\text{int\_op} \leftarrow \text{name}(c).\text{op} \\
&\text{end for}
\end{align*}
\]

For each connector associated with the port, the internal channels \text{int\_sig} and \text{int\_op} are renamed to the channels (operation and signal channels) associated with the connector. For example, the renaming pairs for the port \text{p3} of our examples are as follows.

\[
p3.\text{int\_sig} \leftarrow c\_p1\_p3.\text{sig}, \; p3.\text{int\_op} \leftarrow c\_p1\_p3.\text{op}, \\
p3.\text{int\_sig} \leftarrow c\_p1\_p2.\text{sig}, \; p3.\text{int\_op} \leftarrow c\_p1\_p2.\text{op}
\]

The result of applying the Rule 4.20 to the block \text{System} in Figure 4.2 is the following process.

\[
\text{process} \text{System} = \text{id: ID @ (Machine(id^[\text{\_mk\_token\_machine}])}[]
\]

74
The translation rule that determines the set of allowed events for a part is shown below; it takes a part (that is, an instance of a block that is part of a composite block), and builds the channel set that defines the possible interactions of the part with its environment. This set is built based on the connectors that originate in the part or its ports. The ends of a connector (c.Ends) may be parts or ports, and to identify the appropriate channel, we use the name of the block that types the part, or the name of the port. We assume the existence of an attribute ClassifierName common to parts and ports that contains the block that types the part or the port itself. For instance, for a port p, the attribute p.ClassifierName is p.
Rule 4.25 \( \text{t\_chanset\_part}(p: \text{Part}); \text{chanset} = \)

\[
\{ \\
\text{for each c: p.Connectors do} \\
\text{let x in set c.Ends such that x <> p in} \\
\text{name(x.ClassifierName).op.n.(id^id(p)).(id^id(x))}, \\
\text{name(x.ClassifierName).sig.n.(id^id(p)).(id^id(x))}, \\
\text{name(x.ClassifierName).op.n.(id^id(x)).(id^id(p))}, \\
\text{name(x.ClassifierName).sig.n.(id^id(x)).(id^id(p))}, \\
\text{end for} \\
\text{for each x: p.Ports do} \\
\text{name(x.ClassifierName).op,name(x.ClassifierName).sig} \\
\text{for each c: x.Connectors do} \\
\text{let y in set c.Ends such that y <> x} \\
\text{name(c).op.n.(id^id(x)).(id^id(y))}, \\
\text{name(c).op.n.(id^id(y)).(id^id(x))}, \\
\text{name(c).sig.n.(id^id(x)).(id^id(y))}, \\
\text{name(c).sig.n.(id^id(y)).(id^id(x))}, \\
\text{end for} \\
\text{end for} \\
\text{name(p.ClassifierName).op,name(p.ClassifierName).sig} \\
\text{n: nat \} }
\]

This translation rule takes a part and an identifier, and produces a channel set that specifies the set of events on which the part can synchronise. This set includes:

- for each connector linked to the part, the events associated with the channels \text{op} and \text{sig} of the other end of the connector; in this case, the second and third parameters are the identifiers of the both ends of the connector in any order;

- for each port of the part, the events associated with the channels \text{op} and \text{sig} of the connector are added; these events are restricted to those where the second and third parameters are the identifiers both ends of the connector in any order.

- all the events associated with the channels \text{op} and \text{sig} of the part are in the channel set. These events allow anyone to communicate with the block.

This channel set does not restrict which elements can direct requests to the part or its ports, not does it restrict which elements can receive responses from the part or
its ports. It does, however, restrict the elements to which the part and its ports can
direct requests. A similar restriction exists regarding which elements can respond
to the part and its ports. The intersection of this channel set with the channel
set associated with a connected element determines the allowed communication
between both ends.

In our example, the channel set associated with the part `screen`, which in has
identifier `id[^mk_token(screen)]` is shown below.

```
{| c_p1_p2.op.n.(id^mk_token(screen)).(id^mk_token(machine)),
c_p1_p2.op.n.(id^mk_token(machine)).(id^mk_token(screen)),
c_p1_p2.sig.n.(id^mk_token(screen)).(id^mk_token(machine)),
c_p1_p2.sig.n.(id^mk_token(machine)).(id^mk_token(screen)) |
| n: nat |
|
```

This channel set essentially allows the part `screen` to communicate on the chan-
nels associated with the connector between ports `p1` and `p2`, where the second
and third parameters are the identifiers of the parts `machine` and `screen` in both
orders.

### 4.7 Final remarks

In this chapter, we have presented a behavioural model of SysML blocks that in-
cludes simple and composite blocks, generalisation, association and composition
relations, standard ports and connectors, interfaces, operations, attributes and sig-
nals. To the best of our knowledge, this is the first formalisation of the behavioural
semantics of a comprehensive subset of the block notation. In particular, it is also
unknown to us any treatments of SysML blocks that take into account state ma-
chine diagrams and activity diagrams.

[Gra09] proposes a representation of a restricted subset of SysML block diagrams
in the web ontology language (OWL2). OWL2 is a language for knowledge rep-
resentation based on a description logic. Description logics are subsets of first-
order logic that possess better computational properties (e.g., some description
logics are decidable and others have efficient inference procedures). [DT10] pro-
poses a representation of SysML block diagrams directly in a description logic.
In both cases, block diagrams are restricted to have only associations and simple
blocks.

[GBT11] extends [Gra09] by encoding SysML diagrams into a type theory that
axiomatises block diagram notions of types, properties and operators. Block definition diagrams and internal block diagrams are covered, but dynamic aspects of SysML diagrams are not.

All these works focus on generating a set of axioms that specify a system based on a SysML diagram, and then using the existing techniques for the underlying logic to check properties such as consistency. Although [GB11] describe model refinement as theory refinement (that is, modification of the knowledge base aiming at achieving consistency), it does not elaborate on the topic, and it is not clear what properties are preserved by this notion of refinement.
Chapter 5

State machine diagrams

In this chapter, we present the formalisation of SysML state machine diagrams in the form of a set of translation rules. This semantics covers simple, non-orthogonal and orthogonal composite states (and regions), completion and non-completion transitions (including interlevel transitions), final states, initial, junction, join and fork pseudostates, signal and call events, and deferred events.

The overall structure of our models is shown in Figure 5.1. The model of the state machine diagram is a process formed by the parallel composition of two other processes: controller and internal state machine. Similarly to the processes that model blocks, the processes that model a state machine and its parts are parametrised by an identifier.

The controller process interacts with the environment through the channel `addevent` and with the internal state machine process through the channel `inevent` and `consumed`. This process receives events from the environment (e.g., the block process as described in the previous chapter), and assigns one event at time to be processed by the state machine. This process is formed by an event pool and a queue of deferred events, and models the priorities between normal and deferred events.

The internal state machine process models the behaviour of the state machine when processing a particular event (supplied by the controller); it takes an event from the controller through the channel `inevent` and informs the controller whether the event has been successfully consumed or not, or whether it has been deferred. The state machine itself, and each state, region, final state, transition (including intermediate junction pseudostates), join pseudostate and fork pseudostate (including the transitions connected to them) generate CML actions in the internal state machine process.
In particular, each state produces six actions, each region and final state produces five actions, each transition produces a single action, and each join and fork pseudostate produce a single action. Initial pseudostates do not produce independent actions, contributing to the definition of states and regions. The overall behaviour of the internal state machine process is defined as the parallel composition of these actions, with the internal communications hidden.

The interaction between states and non-completion transitions (i.e., triggerless transitions that are executed when a state execution terminates) is modelled by a two phase communication protocol. In the first phase, the event being treated is transmitted to all transitions and enabled transitions (that is, transitions whose trigger, if not empty, contains the event and whose guard evaluates to true) prepare to execute or be cancelled (if a conflict occurs). Non-enabled transitions simply wait to restart. In this phase, conflicting transitions (multiple enabled transitions from a common state) are non-deterministically cancelled until a single one remains. In the second phase, when all transitions are either waiting to restart or execute, the machine request that all transitions proceed and waits for them to terminate.

The translation rules for state machine diagrams rely mostly on the information available through the SysML meta-model, but some new information needs to be
added by pre-processing the abstract syntax tree. The only pre-processing necessary for state machine diagrams is the conversion of sets of regions into sequences of regions in any order.

In Section [5.1], we present the examples that are used throughout the chapter to illustrate our translation rules. Section [5.2] gives presents the set of translation rules that define the semantics of state machines, and Section [5.3] discusses some of the related work found in the literature.

5.1 Examples

We illustrate the semantics of state machine diagrams through the state machine diagram that specifies the behaviour of the block **Buffer** in Figure [5.2].

![Figure 5.2: Producer consumer problem: state machine diagram](image)

The formal model of the block **Buffer** is described in Section [4.5] and the definitions necessary for the development of the semantics of the associated state machine diagram are repeated here. They are the following.

```cml
class Buffer_types = begin
  types
  public I = rem_I | add_I;
  public O = rem_O | add_O;
  public S = token inv x == false
  public OPS = I | O;
```

81
The class Buffer_types defines combines all the input types in type of operation requests (I) and the output types in the type of operation completions (O). The types of the signals of the block are combined in the type S and the types I and O are combined in the type OPS. The type of identifiers is defined in Section 3.3 by Rule 3.2 as a sequence of tokens.

types
    ID = seq of token

The channels used to communicate with the block are defined by Rule 4.13 and are as follows for our example.

channels
    Buffer.op: nat*ID*ID*OPS
    Buffer.sig: nat*ID*ID*S
    Buffer.addevent: nat*ID*ID*MSG
    Buffer.get_b: ID*ID*seq of token
    Buffer.set_b: ID*ID*seq of token

The first and second channels support the request of operation calls and signals. The third channel provides the means for the block to add events to particular event pools or queues (in our example, this operation allows the block to add an operation request to the event pool of the state machine), the fourth channel allows reading the single block attribute b, and the fifth channel allows writing to this variable. The first three channels have a parameter of type nat, which identifies a particular instance of a request, and two parameters of type ID, which identify the source and target of a particular request. These parameters are necessary due to our use of parallelism as a central modelling strategy. It is worth noting however that at the level of blocks, the use of parallelism also reflect the natural decomposition of SysML models and is particularly relevant in the context of Systems of Systems, where the different constituent systems are necessarily working in parallel with each other. In addition to the parameters mentioned, the first channel communicates a value of an input or output type of an operation (OPS), the second communicates a signal (S), and the thirds communicates a message (MSG), that is, either an operation or a signal. The last two channels communicate a value of the type of the block attribute (seq of token) as well as the identifiers of the source and target of the request.

The state machine that specifies the behaviour of the block Buffer, and which is the focus of our example is shown in Figure 5.3.
The machine specifies the behaviour of the operations `add` and `rem` of the block `Buffer`. It uses deferred events to delay operation requests when those cannot be processed. The operation `add` is deferred when the machine is in the state `full` and the operation `rem` is deferred when the machine is in the state `empty`. No operation is deferred in the state `mid`. The states of this machine represent three possible configurations of the sequence `b`: `empty` when the sequence is empty, `mid` when the sequence is not empty and has less than the maximum (`MAX`) number of items, and `full` when the sequence has the maximum number of items allowed.

When in the state `empty`, the machine successfully processes requests to add elements to the sequence. When in the state `full`, requests to remove elements from the sequence are successfully processed. When in the state `mid` both types of request can be successfully processed.

The operation `add`, when successfully executed, concatenates the parameter `x` to the state component `b`. If before the operation is executed the machine is in the state `mid` and the size of `b` is one less than the maximum allowed, then a transition takes place to the state `full` and the operation is executed. If the machine is in the state `empty`, a call to `add` triggers a transition to `mid`.

The operation `rem`, when successfully executed, removes the first element of the state component `b`; it uses a let statement to define the value of the first element as `x`, updates `b` with the rest of `b`, and returns the recorded value `x`. When there is only one element in the sequence `b` and the machine is in the state `mid`, a request for the operation `rem` prompts a transition to `empty`. When the machine is in the state `full`, a call to `rem` triggers a transition to `mid`. 

---

Figure 5.3: Producer consumer problem: state machine diagram
Since our first example contains only simple states, we use other self-contained examples (that is, examples that do not require the model of an associated block) to illustrate the semantics of language features not covered in the previous example. In particular, we use the example shown in Figure 5.4 and extracted from [FMS11] to illustrate the semantics of orthogonal states and regions. A particularly interesting aspect of this example is the treatment of transition priorities, that is, when multiple transitions in different levels of the hierarchy can be triggered, priorities are used to determine which one is effectively triggered.

This example contains six states, of which **state 1.1.1**, **state 1.1.2**, **state 1.2.1**, **state 1.2.2** and **state 2** are simple states. The only composite state, **state 1**, has two orthogonal regions. The first contains states **state 1.1.1** and **state 1.2.1**, and the second contains states **state 1.2.1** and **state 1.2.2**.

There are six transitions: three of them are initial transitions, and the remaining are all triggered by the signal **sig1**, but with different guards. The transition **t1** from **state 1** to **state 2** has guard \( x < = 0 \), **t2** from **state 1.1.1** to **state 1.1.2** has guard \( x > = 0 \), and **t3** from **state 1.2.1** to **state 1.2.2** has guard \( x > = 1 \).

The behaviour of this chart is as follows. On initialisation, the initial transition leads to **state 1** being entered, and consequently its two regions are also entered. The initial transitions in the regions lead to the states **state 1.1.1** and **state 1.2.1** to be entered. At this point, the machine waits for an event to react. In our example, we will assume that this machine can receive two events: **sig1** and **sig2**. Notice that only **sig1** causes the machine to react.

From its initial configuration, if an event **sig1** is received, depending on the value of the state variable \( x \), specific transitions may be triggered. The deeper a transition is in the state hierarchy, the higher its priority. In our example this manifests
itself in that the transitions $t_2$ and $t_3$ are attempted first, and only if they fail, transition $t_1$ is attempted. That is, when a state receives a signal, it first offers the signal to its substates. If the signal is not consumed by the state, that is, if it does not lead to a transition being enabled, the state offers it to its immediate transitions, or informs its parent that the state itself has not consumed the signal.

Another example of a state machine that is covered by our example is the one presented in Section 2.3 and repeated here for convenience.

![Figure 5.5: Monitored machine: state machine diagram](image)

This state machine belongs to the block Machine, and initially, it is in the state off. If an event signal on is received, it sends a request to the port p1 to print “Turning on”, deactivates the state off and enters the state on.

The state on is entered through a transition path involving a fork pseudostate, which directly activates the states monitor of the two regions (R1 and R2) of the state on. Before the states reached by the fork pseudostate are activated, however, all its ancestors (on, R1 and R2) are first activated. Once the two monitor states are entered the “do activities” start executing, and if both of them terminate the completion transitions from the states monitor to the join pseudostate takes place. In this case, the machine exits the state on and all its substates, sends a request through the port p1 to print, and enters the state fail. Once in the state fail, the machine can only be restarted after a signal event fix has been received. In this case, the state fail exits, a print request is sent to p1 and the state off is entered. If a signal event off is received before both do-activities terminate, they are interrupted, the state on (and its substates) exit, a print request is sent through the port
5.2 Formal semantics

The root translation rule for state machine diagrams is as follows; it takes a state machine and an associated block, and produces a sequence of CML program paragraphs [WCC+12].

**Rule 5.1**

```
t_statemachine(stm: StateMachine,b: Block):
    seq of program paragraph =
    channels
        name(b)_inevent: ID*(ID*name(b)_types'I);
        name(b)_consumed: ID*DL;
        instate, setstate: ID*ID*bool;
    t_internal_statemachine(stm,b)
    t_controller(stm,b)
    process stm_name(stm) = id: ID @ {
        controller_name(stm)(id)
        || {name(b)_inevent, name(b)_consumed} ||
        internal_name(stm)(id^[mk_token(stm)])
    } (||name(b)_inevent, name(b)_consumed||)
```

The first paragraph declares four channels: name(b)_inevent, name(b)_consumed, instate and setstate. These channels allow the state machine to select an event from the event pool, signal the consumption or deferral of a selected event, check the status of a state and set the status of a state. This rule uses two other rules to define two processes. The first models the internal behaviour of the state machine with respect to a selected event, and the second models the process that controls the event pool and the queue of deferred events. Finally, these two processes are composed in parallel to define the process stm_name(stm). The two processes synchronise on the events that allow the machine to select an event, signal its consumption and defer it. Since these actions are all internal to the state machine, they are made internal using hiding (\). The result of applying this rule to our example is as follows.

```
channels
    Buffer_inevent: ID*(ID*Buffer_types'I);
    Buffer_consumed: ID*DL;
```
instate, setstate: ID*ID*bool;
process controller_Buffer = ...
process internal_Buffer = ...
process stm_Buffer = id: ID @ (controller_Buffer(id)
               [\{|Buffer_inevent, Buffer_consumed|\}
               internal_Buffer(id^[mk_token(stm)])
            )\{|Buffer_inevent, Buffer_consumed|\}

The definitions of the processes controller_Buffer and internal_Buffer are given by the rules defined in the following sections.

5.2.1 Event controller

As previously mentioned, the controller is responsible for managing the event pool and the queue of deferred events. The controller is modelled by a CML process produced by the Rule 5.2 below. The process produced by this rule declares two state components: events and deferred. Both state components are sequences of quadruples formed by a natural number that identifies the instance of a request, the identifiers of the source and target of the request, and either a signal or an input operation (output operations are synchronous responses and do not enter event pools). The first sequence models the event pool and the second models the queue of deferred events.
Rule 5.2 \texttt{t\_controller(stm: StateMachine, b: Block):}

\begin{verbatim}
process controller\_name(stm) = id: ID @
begin

    types
    E = nat\*ID\*name(b)\_types\'I | nat\*ID\*name(b)\_types\'S;
    state events: seq of E;
    deferred: seq of E

    functions
    remove: (seq of E) \* nat -> seq of E
    remove(s,i) = s(1,...,i-1)^s(i+1,...,len s)
    pre i <= (len s)

    actions
    TreatDeferredEvents = dcl i: nat := 1 @
    while (i <= len deferred) do
        let ev = deferred(i) in
            name(b)\_inevent?o!ev -> (  
                name(b)\_consumed!o?b:(is\_bool(b)) ->
                deferred := remove(deferred,i)
            []
            name(b)\_consumed.o.<defer> -> i := i+1
        )
    []
    mu X @ name(b).addevent?o!id?e ->
        events := events^[mk_(o,id,e)];X
    @ mu X @ (name(b).addevent?o!id?e ->
        (events := events^[mk_(o,id,e)]);X
    [] ![len events) > 0] &
    let i in set inds events in
        let ev = events(i) in
            name(b)\_inevent?o!ev->
                events := remove(events,ev); (  
                    name(b)\_consumed.o.true ->
                        TreatDeferredEvents; X
                [])name(b)\_consumed.o.false -> X
                []name(b)\_consumed.o.<defer> ->
                    deferred := deferred^[ev];X
            )
    )
end
\end{verbatim}
The process declares the function `remove` that takes a sequence and an index \( i \) (of the sequence) and returns a new sequence identical to the input sequence except that the element in the \( i \)-th position is removed. Additionally, an action called `TreatDeferredEvents` is declared; it traverses the list `deferred`, and sends it to the state machine’s internal process; if the event is consumed it is removed from the list of deferred events, otherwise nothing changes. The recursive action inside the while statement guarantees that whenever an events is ready to be sent to the internal process, the block can also add an event to the the pool of events.

Finally, the main action is a recursive action that allows the block to add new events to the pool, or offers a random event in the event pool `events`. Three possibilities can be observed at this point. If an event is successfully consumed, a change in the state may occur, and the deferred events may become active, therefore they must be attempted. This is accomplished by calling the action `TreatDeferredEvents`. If the event was consumed but not successful, no state change has occurred, thus no deferred event can become active. The last case that needs to be treated is when the event is deferred by the state machine. In this case, the event is simply added to the end of the sequence `deferred`. The controller for our example is as follows.

```plaintext
process controller_Buffer = id: ID @
begin

types
  E = nat*ID*ID*Buffer_types'I | nat*ID*ID*Buffer_types'S;

state
  events: seq of E;
  deferred: seq of E

functions
  remove: (seq of E) * nat -> seq of E
  remove(s,i) ==
    s(1,...,i-1)^s(i+1,...,len s)

pre
  i <= (len s)

actions
  TreatDeferredEvents = dcl i: nat := 1 @
  while (i <= len deferred) do
    let ev = deferred(i) in
    Buffer_inevent?o!ev -> (Buffer_consumed!o?b:(is_bool(b)) ->
      deferred := remove(deferred,i)
      []
      Buffer_consumed.o.<defer> -> i := i+1
    )
    []
    mu X @ Buffer.addevent?o!id?e ->
      events := events^[mk_(o,id,e)]; X
  )
@ mu X @ {
```
Buffer.addevent?o!id?e ->
   events := events^[mk_(o,id,e)]; X
   []
   [(len events) > 0] &
   let i in set inds events in {
      let ev = events(i) in {
         Buffer.inevent?o!ev ->
            events := remove(events,i); {
               Buffer_consumed.o.true ->
                  TreatDeferredEvents; X
               []
               Buffer_consumed.o.false -> X
               []
               Buffer_consumed.o.<defer> ->
                  deferred := deferred^[ev]; X
            }
         }
      }
  }
end

This process declares the type $E$ as the quadruples of natural numbers, source and target identifiers, and signal or input operation values accepted by the state machine (and the block that contains it). It declares the event pool and the queue of deferred events, the function that removes an event from a sequence, and the action that treat the deferred events. Finally, it declares the main action of the process that accepts events from the environment and directs events to the internal state machine process.

5.2.2 Internal state machine

The process that models the internal behaviour of the state machine is defined by the following rule.
Rule 5.3 \texttt{t\_internal\_statemachine}(stm: StateMachine,b:Block):

\begin{verbatim}
process internal\_name(stm) = id: ID @ begin 
values 
  toproregions={ \{id(r) | r in set stm.immediateregions\}};
chansets
  internal = \{|enter,entered,exit,exited,
               enabled,fire,fired,name(b)\_consumed.x,
               name(b)\_inevent.x | x: ID @ x<>id(stm)|};
  t\_actions\_and\_chansets(stm,b)
actions
  t\_machine(stm,b)
@ t\_internal\_main\_action(stm)
end 
chansets
  chanset\_name(stm) = \{|enter.id(stm).x,
                    entered.id(stm).x,fire,fired,
                    name(b)\_inevent.x,name(b)\_consumed.x | 
                    x in set toproregions|}
\end{verbatim}

The process defined by this rule abstracts away the management of events, and restricts itself to the behaviour of the state machine under a particular event. It defines a constant toproregions that identifies the set of the names of the regions directly contained by the state machine.

The set of internal channels internal is defined containing all events used by the internal actions to coordinate their behaviour. The actions that model each element of the state machine are declared along with their associated channel sets. The action that models the top level coordination of the machine is defined by the rule \texttt{t\_machine} presented in Rule 5.4
This action initially enters all the machine’s top regions, and recursively reads an event from the controller, sends the event to each top region in interleaving, waits for each region to answer whether the event was successfully consumed, and if it was consumed by at least one, it signals to the controller that the signal was consumed. If it is not consumed by anyone, but deferred by at least one region, the events is deemed deferred and this is communicated to the controller. The decision of whether to consumed or defer the event is based on the output of the function $\text{DL}_{\text{or}}$ defined in Chapter 3. Finally, it signals the transitions to be fired and recurses. The result of applying Rules 5.3 and 5.4 to our example is as follows.

```plaintext
process internal_Buffer = id: ID @ begin
values
  topregions = \{mk_token(r1)\}
chansets
  internal = \{enter, entered, exit, exited, enabled, fire, fired, Buffer_consumed.x, Buffer_inevent.x \}
  x: ID @ x <> \{mk_token(Buffer)\}

actions
machine_Buffer =
```
This process declares a constant that identifies the top regions and the set of internal channels. It then declares the actions that model the elements of the diagram (e.g., states) as well as the associated channel sets. Next, it declares the action that specifies the coordination of events and transitions. Finally, it completes the process by defining the main action (elided here), and declares the set of internal channels.

The elided parts are completed by calls to other rules; the result of the application of those rules is illustrated together with the rule definition.

The next rule produces all the action declarations for the process that models the internal behaviour of the state machine.
Rule 5.5 \texttt{t\_actions\_and\_chansets(stm: StateMachine, b: Block): seq of program paragraph =}
\begin{description}
\item \texttt{t\_states(stm.states,b)}
\item \texttt{t\_finals(stm.finals,b)}
\item \texttt{t\_regions(stm.regions,b)}
\item \texttt{t\_transitions(stm.transitions,b)}
\item \texttt{t\_comp\_transitions(stm.comptransitions,b)}
\item \texttt{t\_joins(stm.joins,b)}
\item \texttt{t\_forks(stm.forks,b)}
\end{description}

For each state, final state, region, transition, completion transition, join and fork pseudostate, this rule contributes to the declaration of the actions that model the corresponding element. The result of the application of this rule to our example is the composition of the results of the application of the individual rules; they will be illustrated after each rule is defined.

The main action of the process is defined by the next rule. The action produced by this rule is the alphabetised parallel composition of the actions that model each of the components mentioned above, with the set of internal channels hidden.

Rule 5.6 \texttt{t\_internal\_main\_action(stm: StateMachine): action =}
\begin{description}
\item \texttt{define\_alphabetised\_parallel(}
\begin{description}
\item \texttt{\{(state\_name(s),chanset\_name(s))| s in set stm.states\}}
\item \texttt{\{(final\_name(s),chanset\_name(s))| s in set stm.finals\}}
\item \texttt{\{(region\_name(s),chanset\_name(s))| s in set stm.regions\}}
\item \texttt{\{(trans\_name(s),chanset\_name(s))| s in set stm.transitions\}}
\item \texttt{\{(trans\_name(s),chanset\_name(s))| s in set stm.comptransitions\}}
\item \texttt{\{(trans\_name(s),chanset\_name(s))| s in set stm.joins\}}
\item \texttt{\{(trans\_name(s),chanset\_name(s))| s in set stm.forks\}}
\item \texttt{\{(machine\_name(stm),chanset\_name(stm))\}\_internal}
\end{description}
\end{description}

A partial view of the result of the application of this rule is shown below.

\begin{verbatim}
(machine_Buffer
 [chanset_Buffer || chanset_r1 union ...]
 (region_r1
  [chanset_r1 || ...]
  ...
 })\_internal
\end{verbatim}

94
The rule `define_alphabetised_parallel` is defined by the Rule [4.21](#) in Chapter [4](#).

### 5.2.3 State

For the purposes of the translation rules, we distinguish three types of states: simple, composite with single region, and composite with multiple regions (orthogonal). We could consider composite states with any number of regions, but by providing separate translation rules for the cases of single and multiple regions, we obtain simpler models, where an extra action that would be generated by the single region composite state is omitted.

The first translation rule (Rule [5.7](#)) takes a set of states and declares the actions and channel sets associated with each state. Each state in the diagram produces six actions, the final one, `state_name(s)` action, models the complete behaviour of the state.

**Rule 5.7** \( t_{\text{states}}(ss: \text{set of State}, b: \text{Block}): \text{seq of process paragraph} = \)

```plaintext
for s in set ss do
    actions
    \( t_{\text{enter\_state}}(s,b) \)
    \( t_{\text{exit\_state}}(s,b) \)
    \( t_{\text{process\_event}}(s,b) \)
    \( t_{\text{completion\_transitions}}(s,b) \)
    \( t_{\text{status}}(s,b) \)
    state\_name(s) = ( 
        in\_name(s)
        \{|{(instate.id(s).id(s),setstate|)}\}
        off\_name(s)
        \{|{(instate.id(s).id(s),setstate|)}\}
    )
    t_{\text{state\_channels}}(s,b)
end for
```

For each state, six CML actions are declared. The first (off action) models the state while inactive, and the second (on action) models the state while active. The third (a action) is used by the on action to process events, the fourth (c action) is used to model the occurrence of completion events, and the fifth records the status of the state. This action controls access to the status of the state through communications; it is needed because the status can be modified and observed...
internally by the state, but can also be observed externally through the channel \textit{instate}.

Finally, the \texttt{off} action is composed in parallel with an action that records the status of the state to form a \texttt{state} action which models the behaviour of the state (initially inactive). The parallelism in the composition of the \texttt{off} and status actions is necessary because the status of the state must be accessible by both the \texttt{off} action and the environment.

For our example, the generated paragraphs are as follows.

\begin{verbatim}
actions
... state_empty = (
  in_empty
  [null|instate.[mk_token(empty)].[mk_token(empty)],
   setstate]]]
  off_empty
  )\{[instate.[mk_token(empty)].[mk_token(empty)], setstate]}

channels
... actions
... state_mid = (
  in_mid
  [null|instate.[mk_token(mid)].[mk_token(mid)], setstate]]
  off_mid
  )\{[instate.[mk_token(mid)].[mk_token(mid)], setstate]}

channels
... actions
... state_full = (
  in_full
  [null|instate.[mk_token(full)].[mk_token(full)], setstate]]]
  off_full
  )\{[instate.[mk_token(full)].[mk_token(full)], setstate]}

channels
...
\end{verbatim}

These actions are the parallel composition of the state’s \texttt{off} action and its status action, communicating on the channels \textit{instate} and \textit{setstate}. The event of the state asking itself its status and all events that set the status are internalised by the hiding operator.

The \texttt{off} action waits for a request to enter the state, sets the state status as active (\texttt{setstate!name(s)!name(s)!true}), tries to enter its parent state, executes its
entry action (s.entry), indicates the state has been entered, tries to enter its substates, and behaves like the on action.

**Rule 5.8**  
\[
\text{t\_enter\_state}(s: \text{State}, b: \text{Block}): \text{action declaration =}
\]

\[
\begin{align*}
\text{off\_name}(s) &= \text{enter}?o!(\text{id}(s)) \to \text{setstate!}\text{id}(s)!\text{id}(s)!\text{true} \to ( \text{t\_enter\_parent}(s, b); \text{t\_action}(s, b, \text{name}(s), []); \text{entered!o!(s)} \to \text{Skip}; \text{t\_enter\_children}(s, b) ) ; \text{on\_name}(s)
\end{align*}
\]

For the state empty in our example, the action declaration produced by this rule is as follows.

\[
\begin{align*}
\text{off\_empty} &= \text{enter}?o![\text{mk\_token}(\text{empty})] \to \text{setstate![mk\_token(\text{empty})]}!\text{mk\_token(\text{empty})]}!\text{true} \to ( \text{Skip}; \text{Skip}; \text{entered!o![\text{mk\_token(\text{empty})}]} \to \text{Skip}; \text{Skip})
\end{align*}
\]

When entering a parent state or region, the actual steps depend on whether the parent is the machine or not. If it is, then nothing is done. Otherwise, the status of the parent is checked, if it is inactive (\(b = \text{false}\)), the action requests that it is entered, waits for the entering process to finish and terminates. Rules 5.9 and 5.10 formalise these actions.

**Rule 5.9**  
\[
\text{t\_enter\_parent}(s: \text{State}, b: \text{Block}): \text{action =}
\]

\[
\begin{align*}
\text{instate!}\text{id}(s)!\text{id}(s.\text{parent})?b \to ( \\
\text{if } b = \text{false} \\
\text{then } \text{entered!}\text{id}(s)!\text{id}(s.\text{parent}) \to \\
\text{else } \text{Skip}
\end{align*}
\]

provided

1. The parent of \(s\) is not the machine.
Rule 5.10 \[ t_{\text{enter\_parent}}(s: \text{State}, b: \text{Block}): \text{action} = \]

\[ \text{Skip} \]

provided

1. The parent of \( s \) is the machine.

When entering a state, its children must be entered. We distinguish three different situations: the state is simple, the state is non-orthogonal composite, and the state is orthogonal composite. Rules [5.11][5.12] and [5.13] generate the actions that model each case. The first case, for simple states, is trivial; it simply does nothing.

Rule 5.11 \[ t_{\text{enter\_children}}(s: \text{State}, b: \text{Block}): \text{action} = \]

\[ \text{Skip} \]

provided

1. \( s.\text{isComposite} = \text{false} \).

The rule for non-orthogonal composite states generates an action that checks whether one of its substates is active or not. If it is, the action terminates immediately. Otherwise, it executes its initial pseudostate (as defined by the action specified by the Rule [5.58]).

Rule 5.12 \[ t_{\text{enter\_children}}(s: \text{Vertex}, b: \text{Block}): \text{action} = \]

\[ (\text{dcl } b: \text{bool} := \text{false} @
  (\text{for all } x \text{ in set } \{ \text{name}(x) \mid x \text{ in set } s.\text{substates} \}) \text{ do}
    \text{instate}!\text{id}(s)!x?y \rightarrow b := b \text{ or } y
  )
)
\[ \text{if } b = \text{false} \]
\[ \text{then } t_{\text{initial\_pseudostate}}(s, b) \]
\[ \text{else Skip} \]

provided

1. \( (\text{is\_State}(s) \text{ and } s.\text{isComposite} = \text{true} \text{ and } s.\text{isOrthogonal} = \text{false}) \text{ or } \text{is\_Region}(s) \).
Rule 5.13 \texttt{t}\_enter\_children(s:\text{State, b:Block}): action =

\[
||| r: \{ \text{name}(x) \mid x \in \text{set} s.\text{regions} \} \ |
\text{instate}!\text{id}(s)!r?x -> (\|\| \\
\quad \text{if} x = \text{false} \\
\quad \quad \text{then} \text{enter}!\text{id}(s)!r -> \text{entered}!\text{id}(s)!r -> \text{Skip}
\quad \text{else} \text{Skip}
)
\]

provided
1. \text{s.isComposite} = \text{true} and \text{s.isOrthogonal} = \text{true}.

After a state is entered (and after all its children are entered), its “do activity” is executed at the same time as its (non-completion) transitions are offered. Once the “do activity” terminates, any completion transitions leaving the state are executed (\text{c}_.\text{name}(s)). If a transition requests the state to exit, any “do activity” or transition offering still occurring are interrupted and the state is exited. When exiting a state, it is first marked as inactive, then its children (if any) are exited, its exit behaviour (\text{t}_.\text{actions}(s.\text{exit},b.\text{name}(s),[])]) is executed, it signals the end of exiting the state (\text{exited}), and exits the parent (if this is still active). Afterwards, the state behaves as an inactive state (\text{off}_.\text{name}(s)).

Rule 5.14 \texttt{t}\_exit\_state(s:\text{State, b:Block}): action declaration

\[
\text{on}_.\text{name}(s) = \{\|\| \\
\quad (\text{t}_.\text{actions}(s.\text{do},b.\text{name}(s),[]];\text{c}_.\text{name}(s))
\quad \|\| \\
\quad \text{a}_.\text{name}(s)
\}/\text{exit}?o!\text{id}(s) -> \text{setstate}!\text{id}(s)!\text{id}(s)!false -> (\|\| \\
\quad \text{t}_.\text{exit}\_children(s,b);\text{t}_.\text{actions}(s.\text{exit},b.\text{name}(s),[])];
\quad \text{exited}?o!\text{id}(s) -> \text{Skip};\text{t}_.\text{exit}\_parent(s,b)
\}; \text{off}_.\text{name}(s)
\]

The \text{on}_. action for the state \text{empty} is as follows.

\[
\text{on}_.\text{empty} = \{\|\| \\
\quad \text{Skip}; \text{c}_.\text{empty}
\quad \|\| \\
\quad \text{a}_.\text{empty}
\}/\text{exit}?o![\text{mk\_token} (\text{empty})] ->
\]
The Rules 5.15, 5.16 and 5.17 declare an action that models the processing of input events. These rules distinguish between simple, non-orthogonal and orthogonal states. In general, this action is responsible for offering an event (received by the state) to its substates or regions, and its (non-completion) transitions. If either the substates or the transitions consume the event, this is communicated through the channel \texttt{name(b\_consumed)}. If the event is not consumed by either substates or transitions, this is also communicated through \texttt{name(b\_consumed)}.

**Rule 5.15** \( t\_process\_event(s: \text{State}, b: \text{Block}): \text{action declaration} = \)

\[
a_{\text{name}(s)} = \mu X @ \text{name(b\_inevent}\!\!id(s)e \to t\_transition\_offers(s,b); X
\]

provided

1. \text{s.isComposite} = \text{false}.

For the state \textit{empty}, the result of the application of this rule is as follows.

\[
a_{\text{empty}} = \mu X @ \text{Buffer\_inevent}[\text{empty}]\!\!e \to \{
\text{let ts = \{empty\_mid,empty\_rem\} in}
\text{dcl i : nat @ i := 0; (}
\text{||| t : ts @ Buffer\_inevent!t!e \to ene\text{d\text{abled}}!t?b \to \text{Skip})}
\text{|||\{\{\text{enabled}\}\{\{i\}\}\}
\text{for t in set ts do ene\text{d\text{abled}}?x?y \to (if y = true then i := i+1 else Skip)}
\text{)}
\text{);(||| j: \{i,...,i-1\} @ cancel!\text{empty}\!\!x \to \text{Skip};}
\text{if i > 0 then Buffer\_consumed!\text{empty}\!\!true \to \text{Skip}
\text{else (if false then Buffer\_consumed!\text{empty}\!\!<\text{defer} \to \text{Skip}
\text{else Buffer\_consumed!\text{empty}\!\!false \to \text{Skip})} \}
\text{)}
\text{);X}
\]
This action is a recursion that, at each step, receives an event through `Buffer_inevent`, sends the event to all its transitions (recorded in the local value `ts`), and checks if any of the transitions are enabled by the event. If more than one transitions are enabled, it non-deterministically cancels all transitions but one. Finally, it indicates whether or not the events has been consumed by the transitions, and recurses. The vacuous clause in the last if statement is a product of the check for deferred events in a state that does not defer events.

In the case of non-orthogonal states, the action must first offer the event to its active substates, and only offer it to its transitions if the event has not been consumed or deferred by one of its substates.

**Rule 5.16**

```
t_process_event(s: State, b:Block): action declaration =
```

```
a_name(s) = mu X @ name(b)_inevent!id(s)?e -> (let ss = { [] {y | y in set s.region(1).subvertex @ is_Finalstate(y) or is_State(y)} } in name(b)_inevent?x:(x in set ss)!e -> (if y = false then t_transition_offers(s,b) else name(b)_consumed!id(s)!y -> Skip ) )); X
```

**provided**

1. `s.isComposite = true` and `s.isOrthogonal = false`.

Finally, in the case of orthogonal composite states, the event must first be offered to all its regions, and, if any region consumes it, the action terminates. If no regions consume the event, the event is offered to the non-completion transitions leaving the state. The control of the deferral priority is performed by the use of the function `DL_or` and by the action produced by the Rules 5.18 and 5.19.
Rule 5.17 \texttt{t\_process\_event}(s: \texttt{State}, b: \texttt{Block}): \texttt{action declaration} =

\begin{verbatim}
  a_name(s) = mu X @ (let regions = \{id(r)|r in set s.regions\})
                    in name(b)\_inevent!id(s)?e ->(
    dcl aux: bool @ aux := false;
    (||| r: regions @ name(b)\_inevent!r!e ->
      name(b)\_consumed!r?b -> Skip)
    |||{|||{name(b)\_consumed}|}{aux}|
    (for i in set regions do
      name(b)\_consumed?x?y -> aux := DL_or(aux,y))
  );
  if aux = false
    then \texttt{t\_transition\_offers}(s,b)
    else name(b)\_consumed!id(s)!aux -> Skip
); X
\end{verbatim}

\textbf{provided}
1. \texttt{s.isComposite} = true and \texttt{s.isOrthogonal} = true.

The offering of the event to the transitions is modelled by the action generated by the Rules 5.18 and 5.19. These rules distinguish between a state with and without transitions.

For the case of an empty set of transitions, the action checks if the event is in the state’s deferral set. If it is, it communicates \texttt{defer} through the channel \texttt{name(b)\_consumed}, otherwise it communicates \texttt{false}.

Rule 5.18 \texttt{t\_transition\_offers}(s: \texttt{State}, b: \texttt{Block}): \texttt{action} =

\begin{verbatim}
  if t_triggers(s.deferralSet,b)
  then name(b)\_consumed!id(s)!\texttt{<defer>} -> Skip
  else name(b)\_consumed!id(s)!\texttt{false} -> Skip
\end{verbatim}

\textbf{provided}
1. \texttt{card(t: Transition | t.source = s and card(t.trigger) > 0) = 0}.

For the more interesting case of a non-empty set of transitions, the action offers the event to each transition and if none are enabled, it checks if the event
is in the deferral set. If it is, it communicates $\text{<defer>}$ through the channel $\text{name(b)}_{\text{consumed}}$, otherwise it communicates $\text{false}$. If some of the transitions are enabled, the action (non-deterministically) cancels the enabled transitions, until only one transition remains enabled. At this point, the action indicates that it has consumed the event. In this rule, the size of a set is given by $\text{card}$.

**Rule 5.19**

$t\_\text{transition_offers}(s:\text{State}, b:\text{Block})\colon \text{action} =$

\[
\begin{align*}
&\text{let } ts = \{ \lfloor \{ \text{id(t)} \mid \text{Transition } t.\text{source} = s \text{ and } \text{card}(t.\text{trigger}) > 0 \} \rfloor \\
&\quad \text{in } ( \\
&\quad \quad \text{dcl } i \colon \text{nat} \quad @ \quad i := 0; \{ \lfloor \lfloor t \colon ts @ \text{id(b)}_{\text{-inevent!t!e}} \rightarrow \text{enabled!t?b} \rightarrow \text{Skip} \rfloor \lfloor \lfloor \text{enabled!}[i] \rfloor \rfloor \lfloor \lfloor i \rfloor \rfloor \lfloor \lfloor \text{for } t \text{ in set } ts \text{ do} \text{enabled?x?y} \rightarrow (\text{if } y = \text{true then } i := i + 1 \text{ else } \text{Skip}) \rfloor \rfloor \lfloor \lfloor j \colon \{1, \ldots, i-1\} @ \text{cancel!id(s)?x} \rightarrow \text{Skip} \rfloor \rfloor) \lfloor if \ i > 0 \text{ then } \text{name(b)}_{\text{consumed!id(s)?true}} \rightarrow \text{Skip} \lfloor else ( \lfloor if \ t.\text{triggers}(s.\text{deferralSet},b) \text{ then } \text{name(b)}_{\text{consumed!id(s)?<defer>}} \rightarrow \text{Skip} \lfloor else \text{name(b)}_{\text{consumed!id(s)?false}} \rightarrow \text{Skip} \rfloor \rfloor \rfloor \rfloor) \rfloor ) \rfloor \rfloor \rfloor)
\end{align*}
\]

**provided**

1. $s.\text{isComposite} = \text{false}$.
2. $\text{card}(t:\text{Transition} \mid t.\text{source} = s \text{ and } \text{card}(t.\text{trigger}) > 0) > 0$.

The following rules deal with completion transitions stemming from the state. They take into account the nature of the state. For simple states, the Rule 5.20 declares an action that recursively signals, in interleaving, to each completion transition, that the “do activity” of the state has terminated. Since completion transitions must have mutually exclusive guards [Obj11, p. 584], at most one completion transition is enabled at any time, and, therefore, there is no need to cancel completion transitions. The recursion guarantees that if a completion signal is lost by a join pseudostate (due to a potential change of state) it can be recovered (if the state is still active). If the completion signal is not lost, the recursion is interrupted by the triggered transition.
Rule 5.20 \[ t\text{-completion-transitions}(s:\text{State}, b:\text{Block}):\text{action declaration} = \]
\[
c\text{_name}(s) = \]
\[
\text{let } \text{cts} = \{ t | \text{Transition \@ t.source = s and card(t.trigger) = 0} \} \]
\[
in (||| \text{cti: cts @ } \mu X @ \text{completion!cti} \rightarrow X) \]

\text{provided}
\begin{enumerate}
\item s.isComposite = false.
\end{enumerate}

For non-orthogonal composite states, the completion transitions can only be triggered if the “do activity” has terminated and the state has reached a final state. Rule 5.21 treats this case.

Rule 5.21 \[ t\text{-completion-transitions}(s:\text{State, b:\text{Block}}):\text{action declaration} = \]
\[
c\text{_name}(s) = \]
\[
\text{let } \text{cts} = \{ t | \text{Transition \@ t.source = s and card(t.trigger) = 0} \}, \]
\[
\text{fs} = \{ x | x \in \text{set s.region(1).subvertex @ is_Finalstate(x)} \} \]
\[
in \text{final_state!id(s)} \rightarrow (||| \text{cti: cts @ } \mu X @ \text{completion!cti} \rightarrow X) \]

\text{provided}
\begin{enumerate}
\item s.isComposite = true and s.isOrthogonal = false.
\end{enumerate}

For orthogonal composite states, the completion transitions can only be triggered if the “do activity” has terminated and all its regions have reached a final state. Rule 5.22 treats this case.

Rule 5.22 \[ t\text{-completion-transitions}(s:\text{State, b:\text{Block}}):\text{action declaration} = \]
\[
c\text{_name}(s) = \]
\[
\text{let } \text{cts} = \{ t | \text{Transition \@ t.source = s and card(t.trigger) = 0} \} \]
\[
in \text{define_interleaving({t_region_completion(r)| r in set s.regions})}; (||| \text{cti: cts @ } \mu X @ \text{completion!cti} \rightarrow X) \]

\text{provided}
\begin{enumerate}
\item s.isComposite = true and s.isOrthogonal = true.
\end{enumerate}

This rule applies a rule \text{define_interleaving} that takes a set of actions and recursively builds the interleaving of all them.
**Rule 5.23** \texttt{define interleaving}(as: set of action): action =

\begin{verbatim}
if card as = 0 then Skip
else let a in set as in (
    if card as = 1 then a
    else (a ||| define interleaving(as\{a\}))
)
\end{verbatim}

**Rule 5.24** \texttt{t_region_completion}(r:Region,b:Block): action =

\begin{verbatim}
let fs = {
    \{x | x: Finalstate @ x in set r.subvertex\}
} in final_state!id(r) -> Skip
\end{verbatim}

Similarly to the case of entering substates, when exiting substates, three cases must be considered: simple states, non-orthogonal states and orthogonal states. Rules \texttt{5.25} \texttt{5.26} and \texttt{5.27} cover these cases. The first case simply does nothing.

**Rule 5.25** \texttt{t_exit_children}(s:State, b: Block): action =

\begin{verbatim}
Skip
\end{verbatim}

\textbf{provided}

1. \ s.isComposite = false.

In the case of non-orthogonal composite states, the action must first check whether there are any active substates. If there are, the nature of non-orthogonal composite states guarantees that only one such state is active. In this case, the action requests the exiting of one of the substates and waits for it to finish exiting.
Rule 5.26 \( t_{\text{exit}_\text{children}}(s: \text{State}, b: \text{Block}): \text{action} = \)

\[
\begin{align*}
\text{let } ss &= \{ \text{id}(x) \mid x \in \text{set } s.\text{substates} \} \text{ in (} \\
\quad \text{dcl } b : \text{bool} &:= \text{false} @ ( \\
\quad \quad \text{for all } s \in \text{set } ss \text{ do (} \\
\quad \quad \quad \text{instate!id}(s)!s?x &\rightarrow b := b \text{ or } x \\
\quad \quad \); \\
\quad \quad \text{if } b = \text{true} \\
\quad \quad \text{then } \text{exit!id}(s)?(x \in \text{set } ss) &\rightarrow \\
\quad \quad \quad \text{exited!id}(s)!x &\rightarrow \text{Skip} \\
\quad \quad \text{else } \text{Skip} \\
\quad ) \\
\)
\]

provided

1. \( s.\text{isComposite} = \text{true and } s.\text{isOrthogonal} = \text{false}. \)

In the case of orthogonal composite states, the action that exits its subregions simply checks the status of each region, in interleaving, requests that the active ones are exited, and waits for the exiting process to terminate.

Rule 5.27 \( t_{\text{exit}_\text{children}}(s: \text{State}, b: \text{Block}): \text{action} = \)

\[
\begin{align*}
\quad (\big| r: \{ x \mid x \in \text{set } s.\text{regions} \} \text{ @ } \text{instate!id}(s)!r?x &\rightarrow ( \\
\quad \quad \text{if } x = \text{true} \\
\quad \quad \text{then } \text{exit!id}(s)!r &\rightarrow \text{exited!id}(s)!r &\rightarrow \text{Skip} \\
\quad \quad \text{else } \text{Skip} \\
\quad ) \\
\)
\]

provided

1. \( s.\text{isComposite} = \text{true and } s.\text{isOrthogonal} = \text{true}. \)

Rule 5.28 produces the action that records the status of a state, final state or region. The resulting action allows the status to be set or read.
Rule 5.28 \( t\text{\_status}(o: \text{Object}): \text{action declaration} = \)

\[
\text{in\_name}(o) = (\text{dcl } b: \text{bool} := \text{false } @ \\
\text{mu } X @ (\\
\text{instate}\?x!id(o)!b -> X \\
[] \\
\text{setstate}\?x!id(o)!y -> b:=y; X \\
)}
\]

provided

1. is\_State(s) or is\_Finalstate(s) or is\_Region(s).

Finally, the following rule specifies the channel set of allowed communications for the action that models a particular state.

Rule 5.29 \( t\text{\_state\_channels}(s: \text{State}, b:\text{Block}): \text{chanset declaration} = \)

\[
\text{chansets} \\
\text{chanset\_id}(s) = (| \text{instate}\_x.id(s), \text{enter}\_x.id(s), \\
\text{entered}\_x.id(s), \text{enter}\_id(s), \text{entered}\_id(s), \\
\text{exit}\_x.id(s), \text{exited}\_x.id(s), \text{exit}\_id(s), \\
\text{exited}\_id(s), \text{final\_state}\_z, \text{final\_state}\_id(s), \\
\text{name}(b)\_\text{inevent}\_z, \text{name}(b)\_\text{consumed}\_z, \text{name}(b)\_\text{inevent}\_w, \\
\text{enabled}\_w| x: \text{ID}, z: \text{ID}, w: \text{ID}, t: \text{ID} @ \\
(z = \text{id}(s) \text{or} \\
z \text{in set } (\{\text{id}(x) | x \text{ in set s.substates}\}) \text{ or} \\
z \text{in set } (\{\text{id}(x) | x \text{ in set s.regions}\}) \\
\text{and } w \text{ in set } (\{t | t: \text{Transition} @ t.source = s\}) \\
|) \text{ union } t\text{\_readstate\_chanset}(b, \text{name}(s))
\]

The channel set includes:

- all events associated with the channel \text{instate} directed at the state;
- all events associated with the channels \text{enter}, \text{entered}, \text{exit} and \text{exited}, where the first parameter or the second parameter is the state;
- all events associated with the channel \text{final\_state} where the first parameter is the identifier of the state or one of its subregions;
5.2.4 Regions

The translation rules for regions are similar to those for non-orthogonal composite states (5.8, 5.14, 5.16 and 5.21). The rules apply as if the region belongs to a non-orthogonal composite state with no transitions, no entry or exit actions, and no “do activities”. This is a valid because regions are part of composite state, and cannot have behaviours or transitions (whose source is the region).

Regions are anonymous, therefore an appropriate unique name must be generated. The translation rules for the off and on actions associated with a region are given below. Because regions of a composite state can be entered independently of the parent state (via interlevel transitions), it is necessary to check if the parent is active before activating the regions, and, if not, request that it be activated. Since multiple regions can be entered at the same time (via fork pseudostate) and the activation of the parent is not atomic (the request and the marking of the state as active occur in separate communications), it is not enough to have the regions check the parent and request its activation, because two or more regions may check the status of the parent at the same time, observe it as inactive and request its activation. The parent is only prepared to receive one activation request (when inactive) and the model deadlocks if more request are raised.

To deal with this issue, we devise a simple protocol between a composite state and its orthogonal regions. The protocol enforces that only one of the regions checks and possibly requests the activation of the parent, and the remaining regions are responsible for doing the same to each other in an order that allows the propagation of the activation request from any region to the parent. This protocol relies on an ordering of the regions of the state, but this ordering does not affect the observable execution of the regions. It is only necessary to enforce the protocol. This ordering is reflected in the translation rules by the parameter \( \rho \) that identifies a sibling region that precedes \( r \), or the state that contains \( r \) if the regions is the first in the ordering of regions. Notice that any order is valid for the purposes of this protocol, as long as it is fixed, which is achieved by preprocessing the abstract syntax, and converting the sets of regions into sequences of regions.

Regions are compartments of a composite state. They are similar to composite states themselves, except that each region (and potentially one of its substates)
can be active at the same time. Regions support the specification of concurrent behaviour in state machine diagrams.

Each region generates five actions, the last of them, the region action, uses the first four to model the behaviour of the region. Rule [5.30] generates all the actions associated with the regions (of orthogonal states) in the diagram.

**Rule 5.30**

\[
\text{for } r \text{ in set } rs \text{ do}
\]

\[
\text{actions}
\]

\[
\text{t}\_\text{enter}\_\text{region}(r,b)
\]

\[
\text{t}\_\text{exit}\_\text{region}(r,b)
\]

\[
\text{t}\_\text{process}\_\text{event}\_\text{region}(r,b)
\]

\[
\text{t}\_\text{status}(r,b)
\]

\[
\text{region}_\text{name}(r) =
\]

\[
(\text{in}_\text{name}(r) \{ | \{ \text{instate} \_. \text{id}(r) \}. \text{id}(r), \text{setstate} | \} \} | \text{off}_\text{name}(r))
\]

\[
\{ | \text{instate} \_. \text{id}(r) \}. \text{id}(r), \text{setstate} | \}
\]

\[
\text{chansets}
\]

\[
\text{chanset}_\text{name}(r) =
\]

\[
(\text{instate} \_. y \text{id}(r), \text{enter} \_. x \text{id}(r),
\]

\[
\text{entered} \_. x \text{id}(r), \text{enter} \_. x \text{id}(r), \text{entered} \_. x \text{id}(r), \text{exit} \_. x \text{id}(r),
\]

\[
\text{exited} \_. x \text{id}(r), \text{exit} \_. x \text{id}(r), \text{exit} \_. x \text{id}(r), \text{exit} \_. x \text{id}(r), \text{name}(b) \_. \text{inevent} \_. z,
\]

\[
\text{name}(b) \_. \text{consumed} \_. z |
\]

\[
x: \text{ID}, y: \text{ID}, z: \text{ID}, w: \text{ID}, t: \text{ID} @
\]

\[
(z = \text{id}(r) \text{ or } z \text{ in set } \{ | x | x \text{ in set } r\text{.substates} | \})
\]

\[
|) \bigcup \text{t}\_\text{readstate}\_\text{chanset}(b,\text{name}(r))
\]

\[
\text{end for}
\]

**provided**

1. forall r in set rs @ rs.state.isOrthogonal = true

When entering a region, the off action marks the region as active, checks if the parent state is active or not, and, if it is not, the action enters that state. Otherwise, it does nothing. Finally, it signals that it has entered the region, and enters its substates.
Rule 5.31 \( t_{\text{enter\_region}}(r:\text{Region}, b:\text{Block}) \): action declaration =

\[
\begin{align*}
off_{\text{name}}(r) &= \text{enter}?>!\text{id}(r) -> \text{setstate}?>!\text{id}(r)?>!\text{id}(r)?>!\text{true} -> ( \\
&\quad \text{instate}?>!\text{id}(r)?>!\text{id}(r.\text{parent})?>!x -> ( \\
&\quad \quad \text{if} \ x = \text{false} \ \text{then} \ \text{enter}?>!\text{id}(r)?>!\text{id}(r.\text{parent}) -> \\
&\quad \quad \quad \text{entered}?>!\text{id}(r)?>!\text{id}(r.\text{parent}) -> \text{Skip} \\
&\quad \quad \text{else} \ \text{Skip} \\
&\quad ); \ \text{entered}?>!\text{id}(r) -> \text{Skip}; \ t_{\text{enter\_children}}(r); \\
&\); \ on_{\text{name}}(r)
\end{align*}
\]

The result of applying this rule to the top region of state 1 in Figure 5.4 is shown below.

\[
\begin{align*}
off_{r11} &= \text{enter}?>!\{\text{mk\_token}(r11)\} -> \\
&\quad \text{setstate}?>!\{\text{mk\_token}(r11)\}?>!\{\text{mk\_token}(r11)\}?>!\text{true} -> ( \\
&\quad \quad \text{instate}?>!\{\text{mk\_token}(r11)\}?>!\text{s1}?>!x -> ( \\
&\quad \quad \quad \text{if} \ x = \text{false} \ \text{then} \ \text{enter}?>!\{\text{mk\_token}(r11)\}?>!\text{s1} -> \\
&\quad \quad \quad \quad \text{entered}?>!\{\text{mk\_token}(r11)\}?>!\{\text{mk\_token}(s1)\} -> \text{Skip} \\
&\quad \quad \quad \text{else} \ \text{Skip} \\
&\quad ); \ \text{entered}?>!\{\text{mk\_token}(r11)\} -> \text{Skip}; \\
&\quad \text{dcl} \ b : \text{bool} : = \text{false} @ \\
&\quad \quad \text{(for all} \ x \ \text{in} \ \{\{\text{mk\_token}(s111)\},\{\text{mk\_token}(s112)\}\} \\
&\quad \quad \text{do} \ \text{in}?>!\{\text{mk\_token}(r11)\}?>!x?>!y -> \ b := \ b \ \text{or} \ y); \\
&\quad \text{(if} \ b = \text{false} \\
&\quad \quad \text{then} \ \text{enter}?>!\{\text{mk\_token}(r11)\}?>!\{\text{mk\_token}(s111)\} -> \\
&\quad \quad \quad \text{entered}?>!\{\text{mk\_token}(r11)\}?>!\{\text{mk\_token}(s111)\} -> \text{Skip} \\
&\quad \quad \text{else} \ \text{Skip} \\
&\quad ) \\
&\); \ on_{r11}
\end{align*}
\]

Upon receiving a request to activate, off_{r11} requests (via the channel setstate) that r11 is marked as active, checks the status of its parent (s1) and, if it is inactive (b = false), activates it. Next, it signals its completion, checks if any of its substates is active, and if none are, executes its initial junction (by entering state s111) and behaves like on_{r11}. The initial junction is modelled by an action that requests the appropriate substate to be entered (in this case, it is s111), waits for its activation to complete and terminates. The treatment of history junctions is similar; in this case, the execution of the initial junction is dependent on the absence of a history record for the enclosing state or region. If such a record exists, the recorded substate is entered instead.

The channel set associated with region r11 as specified by Rule 5.38 is shown
below.

\[
\text{chanset}_{r11} = \{ \text{enter.}[\text{mk_token}(r11)].[\text{mk_token}(s111)], \\
\text{entered.}[\text{mk_token}(r11)].[\text{mk_token}(s111)], \\
\text{enter.}[\text{mk_token}(r11)].[\text{mk_token}(s112)], \\
\text{entered.}[\text{mk_token}(r11)].[\text{mk_token}(s112)], \\
\text{enter.}[\text{mk_token}(r11)].[\text{mk_token}(s1)], \\
\text{entered.}[\text{mk_token}(r11)].[\text{mk_token}(s1)], \\
\text{exit.}[\text{mk_token}(r11)].[\text{mk_token}(s111)], \\
\text{exited.}[\text{mk_token}(r11)].[\text{mk_token}(s111)], \\
\text{exit.}[\text{mk_token}(r11)].[\text{mk_token}(s112)], \\
\text{exited.}[\text{mk_token}(r11)].[\text{mk_token}(s112)], \\
\text{exit.}[\text{mk_token}(r11)].[\text{mk_token}(s1)], \\
\text{exited.}[\text{mk_token}(r11)].[\text{mk_token}(s1)], \\
\text{in_event.}[\text{mk_token}(r11)], \text{in_event.}[\text{mk_token}(s111)], \\
\text{in_event.}[\text{mk_token}(s112)], \text{Consumed.}[\text{mk_token}(r11)], \\
\text{consumed.}[\text{mk_token}(s111)], \text{consumed.}[\text{mk_token}(s112)] \}
\]

It includes the same events as the channel set of states, except for the events associated with transitions, because regions do not have transitions.

The on action, while waiting for an event to be processed, may exit the region. When exiting a region, the action interrupts the event processing action, exits the substates, signals that it has exited the region and, if the parent state is active, exits that state. Notice that when exiting the regions, the ordering between the regions and the parent state is taken inverted.

**Rule 5.32** \( t\text{–exit\_region}(r:\text{Region}, b:\text{Block}): \text{action declaration} = \)

\[
\text{on\_name}(r) = \text{a\_name}(r) /\ exit?o!id(r) -> \text{setstate!id(r)}!id(r)!false -> ( \text{t\_exit\_children}(r); \text{exited!o!id(r)} -> \text{Skip}; \\
\text{instate!id(r)}!id(r.parent)?x -> ( \text{if} x = \text{true then} \text{exit!id(r)}!id(r.parent) -> \text{exited!id(r)}!id(r.parent) -> \text{Skip} \\
\text{else} \text{Skip} \}
\}
\text{); off\_name}(r)
\]

A region processes an event by communicating it to the active substates, waiting for a response on\text{name}(b)\_consumed, and communicating on the same channel the response received from the active substate.
Rule 5.33 $t_{\text{process\_event\_region}}(r: \text{Region}, b: \text{Block})$: action declaration =

\[
\begin{align*}
\text{a\_name}(r) &= \{ \text{let } ss = \{ \{ x | x \in \text{set } r.\text{substates} \} \} \text{ in } \nonumber \\
&\quad \text{name}(b)\_\text{inevent}!\text{id}(r)?e \rightarrow \text{name}(b)\_\text{inevent}?x:(x \in \text{set } ss)!e \rightarrow \nonumber \\
&\quad \text{name}(b)\_\text{consumed}.x?y \rightarrow \text{name}(b)\_\text{consumed}!\text{id}(r)!y \rightarrow \text{Skip} \nonumber \\
&\}; \text{a\_name}(r) \nonumber
\end{align*}
\]

There is no action associated with a region.

5.2.5 Final states

Each final state produces five actions. The first action specifies the behaviour of the final state when inactive, the second models the final state when active, the third models the processing of events when the final state is active and the fourth records that status of the final state. These actions are used to form the final_name(f) action that models the behaviour of a final state. Rule 5.34 declares all the actions associated with the final states of the diagram.

Rule 5.34 $t_{\text{finals}}(fs: \text{set of Finalstate}, b: \text{Block})$: seq of process paragraph =

\[
\begin{align*}
\text{for } f \in \text{set } fs \text{ do } \nonumber \\
\quad \text{actions } \nonumber \\
\quad \quad t_{\text{enter\_final\_pseudostate}}(f, b) \nonumber \\
\quad \quad t_{\text{exit\_final\_pseudostate}}(f, b) \nonumber \\
\quad \quad t_{\text{process\_event\_final\_pseudostate}}(f, b) \nonumber \\
\quad \quad t_{\text{status}}(f, b) \nonumber \\
\quad \quad \text{final\_name}(f) = \nonumber \\
\quad \quad \quad \{ \{\text{in\_name}(f)[\{\{\text{instate.}\text{id}(f).\text{id}(f), \text{setstate}\}\}\} \}\ off\_\text{name}(f) \nonumber \\
\quad \quad \quad \\\\\\\\\\\\\\\{\{\text{instate.}\text{id}(f).\text{id}(f), \text{setstate}\}\}\} \nonumber \\
\quad \quad \quad \text{chansets } \nonumber \\
\quad \quad \quad \quad \text{chanset\_name}(f) = \{ |\text{enter}.x.\text{id}(f),$ enter.\text{id}(f), \text{entered}.x.\text{id}(f),$ enter.\text{id}(f), \text{entered}.x.\text{id}(f), \nonumber \\
\quad \quad \quad \quad \text{exit}.x.\text{id}(f),$ exited.\text{id}(f), \text{exit}.x.\text{id}(f), \text{exit}.x.\text{id}(f), \nonumber \\
\quad \quad \quad \quad \text{name}(b)\_\text{inevent}.z,$ name(b)\_\text{consumed}.z \nonumber \\
\quad \quad \quad \quad \quad | x: \text{ID}, y: \text{ID}, z: \text{ID}, w: \text{ID}, t: \text{ID} @ z = \text{id}(f)\} \nonumber \\
\quad \quad \quad \quad \text{union } t_{\text{readstate\_chanset}}(b, \text{name}(f)) \nonumber \\
\quad \quad \text{end for} \nonumber
\end{align*}
\]

Unlike pseudostates, a final state is modelled as a separate action. It behaves
similarly to a simple state, but is simpler. It is initially inactive, and offers to communicate on the channels enter and entered. Additionally, it enters the parent state if necessary.

**Rule 5.35** \( t_{\text{enter\_final\_state}}(s: \text{Finalstate}, b: \text{Block}) \): action declaration =

\[
\text{off}\_\text{name}(s) = \text{enter}\?o!id(s) \rightarrow \text{setState}\!id(s)!id(s)\text{true} \rightarrow \\
\quad \_\_\_\text{enter\_parent}(s); \; \text{entered}\!o!id(s) \rightarrow \text{Skip}; \; \text{on}\_\text{name}(s)
\]

Upon entering, the action \( t_{\text{exit}} \) offers (once) a communication on the channel final_state of the identifier of its parent (state or region), and processes any incoming event. This action may be at any point interrupted by a request (from its parent) to exit on the channel exit (followed by a communication on exited). In this case, the final state becomes inactive.

**Rule 5.36** \( t_{\text{exit\_final\_state}}(s: \text{Finalstate}, b: \text{Block}) \): action declaration =

\[
\text{on}\_\text{name}(s) = ( \\
\quad \text{final\_state}\!id(s,\text{parent}) \rightarrow \text{Skip} \\
\quad \| \| \\
\quad \text{a}\_\text{name}(s) \\
\quad )/\text{exit}\?o!id(s) \rightarrow \text{setState}\!id(s)!id(s)\text{false} \rightarrow \\
\quad \text{exited}\!o!id(s) \rightarrow \text{off}\_\text{name}(s)
\]

A final state is not affected by events. However, since they are treated as substates of a composite state or region, they must accept input events and respond to consumption checks. The \( a \) action of a final state models this behaviour; it iteratively receives events and declares them as not consumed.

**Rule 5.37** \( t_{\text{process\_event\_final\_pseudostate}}(s: \text{Finalstate}, b: \text{Block}) \): action declaration =

\[
\text{a}\_\text{name}(s) = \text{name}(b)\_\text{inevent}\!id(s)\?e \rightarrow \\
\quad \text{name}(b)\_\text{consumed}\!id(s)\text{false} \rightarrow \text{a}\_\text{name}(s)
\]

There is no \( c \) action associated with final states.
5.2.6 Non-completion transitions

Only transitions that start in a state are modelled as a separate action in our translation. Transitions leaving pseudostates (e.g., junctions) are considered part of the transition that reaches that pseudostate. In this section, we present the rules that support the translation of transitions and pseudostates, as well as the rule that generates the actions that models transition flows (sets of transitions connected by pseudostates) starting in states. The following rule declares the actions and channel sets associated with the non-completion transitions that start on a state and do not reach fork or join pseudostates.

**Rule 5.38**  
\[
t\_transitions(ts: \text{set of Transition}; b: \text{Block}): \text{seq of process paragraph} =
\]
for \( t \) in set \( ts \) do
  if \( \text{card}(t.\text{trigger}) > 0 \) and \( t.\text{source} \) in set \( \text{State} \) and
      \( t.\text{target} \) in set \( \text{State} \cup \{x: \text{Pseudostate} | x.\text{kind} \in \{\#\text{junction}\}\} \)
  then
      \text{actions} \quad \text{t\_from\_state}(t, b)
      \text{chansets} \quad \text{chanset\_name}(t) = \{|\text{fire, fired, cancel.} x.\text{id}(t), \text{enter.} x.\text{id}(t),
          \text{entered.} x.\text{id}(t), \text{exit.} x.\text{id}(t), \text{exited.} x.\text{id}(t),
          \text{name(b)}\_\text{inevent.} x.\text{id}(t), \text{enabled.} x.\text{id}(t) | x: \text{ID} |\}
          \text{union} \quad \text{t\_readstate\_chanset}(b, \text{name}(t))
  end if
end for

For each transition that starts in a state, reaches a junction, another state or a final state, and has a trigger, this rule generates an action by calling Rule 5.39 and a channel set composed of:

- the channels \text{fire} and \text{fired} that control the execution of transitions;
- the events associated with the channel \text{cancel}, where the cancellation request is directed at the transition;
- the events associated with the channels \text{enter}, \text{entered}, \text{exit} and \text{exited}, where the first parameter is the transition identifier; and
- the events associated with the channels \text{in\_event} and \text{enabled}, where the parameter is the identifier of the transitions.
Rule 5.39 generates the transaction action associated with a transition that starts in a state. This action waits for an event, and evaluates the trigger of the transitions. If the event is not in the trigger, it indicates that the transition is not enabled, and waits to synchronise on the channels fire and fired before recursing. It is necessary to restart the transition, so that changes in the state are available to the transition even when it is not executed.

If the event is one of the triggers of the transition, a local copy of the state is declared and the guard of the transition is evaluated. If the guard is false, the action indicates that the transition is not enabled, synchronises on fire and fired and recurses. If the guard is true, the action proceeds to the target of the transition, which is specified by the Rule 5.44 and recurses.

If the source state of a transition is not active, it does not propagate the event to the transition. For this reason, it is possible to restart the transitions (through the channels fire and fired) while waiting for an event.

Rule 5.39

t_from_state(t:Transition, b: Block): action declaration =

\[
\begin{align*}
\text{trans}_\text{name}(t) &= ( \\
\quad \text{name}(b) &\_\text{inevent!}\_\text{id}(t) \_\text{?}e \rightarrow ( \\
\quad \quad \text{if} \ t\_\text{triggers}(t\_\text{trigger}, b) \\
\quad \quad \quad \text{then} \ t\_\text{statecopy}(b, \text{name}(t)); \ ( \\
\quad \quad \quad \quad \text{if} \ t\_\text{guard} \\
\quad \quad \quad \quad \quad \text{then} \ t\_\text{to_vertex}(t\_\text{target}, t\_\text{source}, [t\_\text{effect}, \text{name}(t), t\_\text{trigger}, b) \\
\quad \quad \quad \quad \text{else} \ enabled!\_\text{id}(t)!false \rightarrow fire \rightarrow fired \rightarrow Skip \\
\quad \quad \quad ) ) \\
\quad \quad \text{else} \ enabled!\_\text{id}(t)!false \rightarrow fire \rightarrow fired \rightarrow Skip \\
\quad ) ) \\
\quad fire \rightarrow fired \rightarrow Skip \\
); \ text{name}(t)
\end{align*}
\]

provided

1. t.source in set State
2. t.target in set State or (t.target in set Pseudostate => (t.target).kind = #junction)
3. card(t.trigger) > 0
4. t.guard is written in CML.

This rule uses a number of auxiliary rules. The first checks if the event is in the set of triggers of the transition, the second creates a local copy of the state and reads
the state to the local copy, and the third specifies the behaviour of the transition
depending on the type of the target: state (including final states) or junction.

Rule 5.40 \( t_{\text{statecopy}}(b:\text{Block}, n:\text{Name}): \text{action} = \)

\[
\begin{align*}
\text{dc1} & \{ \text{name}(a) : t_{\text{types}}(a.\text{type}) \mid a \in \text{set } b.\text{Attributes} \} \\
&t_{\text{readstate}}(b.\text{Attributes}, b, n)
\end{align*}
\]

Rule 5.41 \( t_{\text{readstate}}(a:\text{set of Attribute}, b:\text{Block}, n:\text{Name}): \text{action} = \)

\[
\begin{align*}
\text{let } a & \in \text{set } a \text{ in} \\
& (\{ \text{name}(b).\text{get}_\text{name}(a).[\text{mk_token}(n)].\text{id}\} \rightarrow \text{name}(a) := x) \\
& \{ \{ \text{name}(x) \mid x \in \text{set } a\backslash\{a\}\} \} \\
t_{\text{readstate}}(a \backslash\{a\}, b, n)
\end{align*}
\]

Rule 5.42 \( t_{\text{readstate_chanset}}(b:\text{Block}, n:\text{Name}): \text{expression} = \)

\[
\{ \{ \text{name}(b).\text{get}_\text{name}(a).[\text{mk_token}(n)].\text{id}, \\
\text{name}(b).\text{set}_\text{name}(a).[\text{mk_token}(n)].\text{id} \mid a \in \text{set } b.\text{Attributes} \} \}
\]

Rule 5.43 \( t_{\text{triggers}}(\text{trs}:\text{seq of Trigger}, b:\text{Block}): \text{expression} = \)

\[
\begin{align*}
\text{if } \text{trs.size()} > 0 & \text{ then} \\
& \{ \{ \text{is}_\text{name}(b).\text{types}'\text{name}(t).\text{I}(e.\#3) \mid t \in \text{trs} @ t \text{ is Operation} \} \\
& \cup \{ \{ \text{is}_\text{name}(b).\text{types}'\text{name}(t).\text{I}(e.\#3) \mid t \in \text{trs} @ t \text{ is Operation} \} \\
& \text{else } \text{false}
\end{align*}
\]

The decision on how to translate the target of the transition is made by Rule [5.44]
Notice that Rule [5.39] only applies to transitions that have at least one trigger.
Transitions without triggers are completion transitions and are treated differently
as described by the Rule [5.49]. The result of applying this rule to the transition
from state **empty** to state **mid** triggered by the operation **add** is shown be-

\[
\begin{align*}
\text{trans_empty_mid} = \\
& \{ \text{Buffer_inevent?[\text{mk_token}('empty_mid')]?!e } \rightarrow \\
\end{align*}
\]
if is_Buffer_types 'add_I(e.#3)
then (dcl b: seq of Item @
  Buffer.get_b ![mk_token(empty_mid)].id?x -> b:=x;
  if true
  then ...
  else enabled ![mk_token(empty_mid)].false ->
    fire -> fired -> Skip
))
else enabled ![mk_token(empty_mid)].false ->
  fire -> fired -> Skip
)
[]
fire -> fired -> Skip
}; trans_empty_mid

The elided portion of trans_empty_mid is completed by the actions defined by
the next few rules. The rule t_to_vertex decides based on the type of the parameter
target how to proceed with the translation. If target is a state, then it applies the
Rule 5.45. If target is a junction, it applies the Rule 5.46. The parameter as is used
to accumulate the behaviours associated with transitions in a path, so that they can
be executed when a state is reached.

Rule 5.44  t_to_vertex(s: Vertex; as: seq of Action; n: Name, trs: seq of Trigger,
             b: Block): action =

  if is_State(t)
  then t_to_state(t, s, as, n, trs, b)
  else t_to_junction(t, s, as, n, trs, b)
  end if

  provided
  1. is_State(t) or (is_Pseudostate(t) and t.kind = #junction)

If the target vertex of a transition is a state, an enabled path has been formed
between two states. In this case, the Rule 5.45 generates the action that models
the execution of an enabled transition. It first communicates that the transition
is enabled through the channel enabled, then waits for a synchronisation on the
channels fire or cancel. If the second synchronisation occurs, the action ter-
m inates, and the trans action recurses. Otherwise, it exits the active substates
of the least common ancestor of the source and target states (of the path that has
been formed), waits for this state to finish exiting, executes the accumulated ef-
facts of the transition path, enters the substate of the least common ancestor of the
source and target that contains the target state, waits for that state to finish entering, synchronises on the channel fired to indicate that the transition has finished executing, and terminates.

**Rule 5.45**

\[
t_{\text{to state}}(t: \text{Vertex}, s: \text{Vertex}, \text{as: seq of Action}, n: \text{Name}, \text{trs: seq of Trigger}, b: \text{Block}): \text{action} =
\]

\[
\text{enabled!}[\text{mk_token}(n)]!\text{true} \rightarrow \\
\text{let } ss = \{ \text{id}(x) \mid x \text{ in set least_common_ancestor}(s, t).\text{substates} \} \\
\text{in} \\
\text{fire} \rightarrow \text{exit!}[\text{mk_token}(n)]?x:(x \text{ in set } ss) \rightarrow \\
\text{exited!}[\text{mk_token}(n)]!x \rightarrow \\
\text{t_actions}(\text{as}, b, n, \text{trs}; \text{enter!}[\text{mk_token}(n)]!\text{id}(t) \rightarrow \\
\text{entered!}[\text{mk_token}(n)]!\text{id}(t) \rightarrow \text{fired} \rightarrow \text{Skip} \\
[\] \\
\text{cancel?x!}[\text{mk_token}(n)] \rightarrow \text{fire} \rightarrow \text{fired} \rightarrow \text{Skip}
\]

provided

1. \text{is State}(t).

The omitted portion of trans_empty_mid is ultimately produced by the previous rule, and is shown below.

\[
\text{enabled!}[\text{mk_token}(\text{empty_mid})]!\text{true} \rightarrow \\
\text{let } ss = \{ \text{r1} \} \\
\text{in} \\
\text{fire} \rightarrow \text{exit!}[\text{mk_token}(\text{empty_mid})]?x:(x \text{ in set } ss) \rightarrow \\
\text{exited!}[\text{mk_token}(\text{empty_mid})]!x \rightarrow \\
\text{...}; \\
\text{entered!}[\text{mk_token}(\text{empty_mid})]!\text{mk_token}(\text{mid}) \rightarrow \\
\text{entered!}[\text{mk_token}(\text{empty_mid})]!\text{mk_token}(\text{mid}) \rightarrow \\
\text{fired} \rightarrow \text{Skip} \\
[\] \\
\text{cancel?x!}[\text{mk_token}(\text{empty_mid})] \rightarrow \text{fire} \rightarrow \text{fired} \rightarrow \text{Skip}
\]

The action still omitted above is obtained by translating the transition action \(b := b^\langle x \rangle\). The rules that support the translation of actions are presented in Section 5.2.10. The omitted portion of this example is shown in that section.

If the target of a transition is a junction pseudostate, the translation Rule 5.46 generates a non deterministic conditional statement where each branch is the result of applying the Rule 5.47 to each transition originating in that junction. An ad-
ditional branch guarded by `else` communicates that the transition is not enabled and waits for an opportunity to restart as indicated via synchronisations on `fire` and `fired`.

**Rule 5.46**

```
let ts = x | x: Transition @ x.source = t in
  if | {t_from_junction(x,s,as,n,trs,b) | x in set ts} |
      else -> enabled! [mk_token(n)]!false ->
                       fire -> fired -> Skip
  fi
```

provided

1. target in set \{x:Pseudostate | x.kind = #junction\}.

**Rule 5.47**

```
t_from_junction(t: Transition, s: Vertex, as: seq of Action, n: Name, trs: seq of Trigger, b: Block): expression -> action =
  t.guard -> t_to_vertex(t.target,s,as^[t.effect],n,trs,b)
```

provided

1. t.source in set \{x: Pseudostate | x.kind = #junction \}.
2. t.target in set State union \{x: Pseudostate | x.kind = #junction\}
3. t.guard is written in CML.

The parameters `source` and `name` used in the last three rules are necessary to keep track of the transition that originated the action `trans_n`. This is important because an action generated from a transition interacts with other components by communicating its identifier, which in the case of a transition flow is the identifier of the transition that originates the flow.
5.2.7 Completion transitions

The translation rules that generate the action that models a completion transition are similar to those for non-completion transitions. The main differences are:

- They do not process events explicitly.
- They do not synchronise on fire and fired.
- They do not need to be cancelled.

Accordingly, for completion transitions, the following rule declares the associated actions.

**Rule 5.48**
\[
\text{t_comp_transitions}(ts: \text{set of Transition}; b: \text{Block}): \text{seq of process paragraph} = \\
\text{for } t \text{ in set } ts \text{ do} \\
\quad \text{t_comp_from_state}(t,b) \\
\quad \text{chansets} \\
\quad \text{chanset_name}(t) = \{ | enter.id(t), entered.id(t), exit.id(t), \\
\quad \quad \text{exited.id(t), completion.id(t)} \} \\
\quad \text{union } \text{t_readstate_chanset}(b, \text{name}(t)) \\
\text{end for} \\
\text{provided} \\
\quad 1. \text{forall } t : ts @ \text{card}(t.\text{trigger}) = 0 \text{ and } t.\text{source} \text{ in set } \text{State} \\
\text{and } t.\text{target} \text{ in set } \text{State} \text{ union } \text{Finalstate} \text{ union } \{x: \text{Pseudostate} | x.\text{kind} = \#\text{junction}\}. \\
\]

**Rule 5.49**
\[
\text{t_comp_from_state}(t: \text{Transition}; b: \text{Block}): \text{action declaration} = \\
\quad \text{trans_name}(t) = \text{completion!}\text{id(t)} \rightarrow \text{t_statecopy}(b, \text{name}(t)); \{ \\
\quad \quad \text{if } t.\text{guard} \\
\quad \quad \text{then } \text{t_comp_to_vertex}(t.\text{target}, t.\text{source}, [t.\text{effect}], \text{name}(t)) \\
\quad \quad \text{else Skip} \\
\quad \} ; \text{trans_name}(t) \\
\text{provided} \\
\quad 1. \text{t.source} \text{ in set State and} \\
\quad \quad (t.\text{target} \text{ in set State or } (t.\text{target} \text{ in set } \text{Pseudostate} \Rightarrow (t.\text{target}).\text{kind} = \#\text{junction})., \\
\quad 2. \text{card}(t.\text{trigger}) > 0 \\
\quad 3. \text{t.guard} \text{ is written in CML.} \\
\]
Rule 5.50 \( t\text{\_}comp\text{\_}to\text{\_}vertex(t;\text{Vertex}, s;\text{Vertex}, as;\text{seq of Action}, n;\text{Name}, b;\text{Block}): \text{action} = \)

\[
\text{if is\_State}(t) \\
\text{then } t\text{\_}comp\text{\_}to\text{\_}state(t,s,as,n,b) \\
\text{else } t\text{\_}comp\text{\_}to\text{\_}junction(t,s,as,n,b) \\
\text{end if}
\]

provided
1. is\_State(t) or (is\_Pseudostate(t) and t.kind = #junction)

Rule 5.51 \( t\text{\_}comp\text{\_}to\text{\_}state(t;\text{Vertex}, s;\text{Vertex}, as;\text{seq of Action}, n;\text{Name}, b;\text{Block}): \text{action} = \)

\[
\text{let } ss = \{ id(x) | x \text{ in set least\_common\_ancestor}(s,t).\text{substates} \} \\
\text{in } \text{exit}! [\text{mk\_token}(n)] !\text{?x:}(x \text{ in set ss}) \rightarrow \text{exited}! [\text{mk\_token}(n)] !x \rightarrow \\
\text{t\_actions}(as,b,n,[]) \;
\text{enter}! [\text{mk\_token}(n)] !\text{id}(t) \rightarrow \text{entered}! [\text{mk\_token}(n)] !\text{id}(t) \rightarrow a\_n
\]

provided
1. is\_State(t).

Rule 5.52 \( t\text{\_}comp\text{\_}to\text{\_}junction(t;\text{Vertex}; s;\text{Vertex}; as;\text{seq of Action}; n;\text{Name}; b;\text{Block}): \text{action} = \)

\[
\text{let ts = x | x; Transition \# x.source = t in} \\
\text{if } \{ \text{t\_comp\text{\_}from\text{\_}junction}(x,s,as,n,b) | x \text{ in set ts} \} \\
\text{else } \rightarrow a\_n
\]

provided
1. t in set \{x:Pseudostate | x.kind = #junction\}

Rule 5.53 \( t\text{\_}comp\text{\_}from\text{\_}junction(t;\text{Transition}; s;\text{Vertex}; as;\text{seq of Action}; n;\text{Name}; b;\text{Block}): \text{expression} \rightarrow \text{action} = \)

\[
t\text{\_guard} \rightarrow t\text{\_}comp\text{\_}to\text{\_}vertex(t.target,s,as^{[t\text{\_}effect]},n,b)
\]

provided
1. t\_guard is written in CML.
2. t\_source in set \{x: Pseudostate | x.kind = #junction \}.
3. t\_target in set State union \{x: Pseudostate | x.kind = #junction\}.
5.2.8 Join and Fork pseudostates

As already explained, join and fork pseudostates are used to converge two or more transitions from different regions of an orthogonal composite state, or, conversely, to fork a transition into two or more transitions that simultaneously reach different regions of an orthogonal composite state.

A number of restrictions apply to this kind of pseudostates. In particular, a fork pseudostate must have exactly one transition reaching it, and at least two transitions leaving it. Similarly, join pseudostates must be reached by at least two transitions, and have exactly one transition leaving them [Obj11, p. 558]. Transitions leaving a fork pseudostate or reaching a join pseudostate must, respectively, reach or leave states contained in different regions of an orthogonal composite state [Obj11, p. 558]. Additionally, transitions reaching or leaving a join pseudostate, and transitions leaving a fork pseudostate cannot have triggers or guards [Obj11, p. 582].

We require that the transition reaching a fork pseudostate or leaving a join pseudostate must, respectively, originate or reach a state, not another pseudostate.

The action that models a join pseudostate waits for completion events from the source states. This is possible because transitions reaching a join pseudostate cannot have triggers or guards. If non-completion transitions in the diagram are executed (via synchronisation on fire and fired) before all transitions receive a completion event, the action stops waiting (interrupted) and recurses. It then executes (if possible) the outgoing transitions, exiting the relevant states and entering the target of the join.

The action that models a fork pseudostate is similar, but it can have guards and triggers in the transition reaching it. It behaves similarly to a transition starting in a state, except that the actions in the outgoing transitions are executed in parallel, and the target states of the outgoing transitions are entered in parallel.

The first two rules declare the actions associated with join and fork pseudostates.
Rule 5.54 \( t_{\text{joins}}(js: \text{set of Pseudostate}, b: \text{Block}): \text{seq of process paragraph} = \)

\[
\text{for } j \text{ in set } js \text{ do } \\
\quad t_{\text{join}}(j) \\
\quad \text{chansets} \\
\quad \text{chanset\_name}(j) = \{|\text{fire, fired, enter.\_id(j), entered.\_id(j)}, \text{exit.\_id(j), exited.\_id(j), completion.x} | x: \text{ID} \@ x \text{ in set } \left\{ \{t|t:\text{Transition @ t.target = j}\} \right\} \} \\
\quad \text{union } t_{\text{readstate\_chanset}}(b, \text{name}(j)) \\
\end{for}
\]

\text{provided}

1. \( js \subset \{x: \text{Pseudostate \mid x.kind = #join}\} \)

For each join pseudostate in the diagram, the action that models it is declared and a channel set is declared. This channel set contains the same events as the channel sets for completion transitions, except that the channels \text{fire} and \text{fired} are also included.

Rule 5.55 \( t_{\text{forks}}(fs: \text{set of Pseudostate}; b: \text{Block}): \text{seq of process paragraph} = \)

\[
\text{for } f \text{ in set } fs \text{ do } \\
\quad \text{let } t = \text{iota } t: \text{Transition @ t.target = f in } \\
\quad t_{\text{fork}}(f) \\
\quad \text{chansets} \\
\quad \text{chanset\_name}(j) = \{|\text{fire, fired, enter.\_id(j), entered.\_id(j)}, \text{exit.\_id(j), exited.\_id(j), } \text{name(b)\_inevent.\_id(t), cancel.x.\_id(t), enabled.\_id(t) | x: ID} | x: \text{ID} \} \text{ union } t_{\text{readstate\_chanset}}(b, \text{name}(f)) \\
\end{for}
\]

\text{provided}

1. \( fs \subset \{x: \text{Pseudostate \mid x.kind = #fork}\} \)

Similarly for each fork pseudostate in the diagram, this rule takes the unique transition that reaches the fork pseudostate using the \text{iota} operator, and declares the action that models it and the associated channel set. Unlike the case of the join pseudostate, the channel set for a fork pseudostate is the same as that of a non-completion transition.

The following rule generates the action that models a join pseudostate.
Rule 5.56  \( t_{\text{join}}(j:\text{Pseudostate}, \ b: \text{Block}); \) action declaration =

\[
\begin{align*}
\text{let} \quad \text{outt} &= \ \text{iota} \ t: \text{Transition} \ @ \ t.\text{source} = j \ \text{in} \\
\text{trans_name}(j) &= \ \mu \ X \ X \ @ \ { (} \ \text{let} \ t s = \ \{ \ \text{id}(t) | t: \text{Transition} \ @ \ t.\text{target} = j \} \ \text{in} \ ( \\
&& \ | | | \ t: ts @ \text{completion}!t -> \text{Skip}) \text{/} \\
&& \ | | | \ \text{fire} -> \text{fired} -> \text{trans_name}(j) \\
\text{)}; \\
\text{t_statecopy}(b, \text{name}(t)); \ ( \\
\text{if} \ \text{outt.guard} \ \text{then} \ \text{exit}!\text{id}(\text{outt})!\text{id}() -> \text{exited}!\text{id}(\text{outt})!\text{id}() -> \\
\quad \ | | | \ { \text{t_action}(t.\text{effect}, b, \text{name}(\text{outt}), []) | \ t \ \text{in} \ \text{set} \ \text{Transition} @ \ \text{t.\text{target}} = j \} \}; \\
\text{t_action}(outt.\text{effect}, b, \text{name}(outt), []); \\
\text{enter}!\text{id}(\text{outt})!\text{id}(outt.\text{target}) -> \omega \\
\text{else} \ \text{fire} -> \text{fired} -> \text{Skip} \\
\text{);} \ X
\end{align*}
\]

where

1. \( \text{src} \) is the the orthogonal composite state that contains the regions from which the transitions originate.

provided

1. \( j \ \text{in} \ \text{set} \ { x: \text{Pseudostate} | x.\text{kind} = \#\text{join} } \)

For a join pseudostate, the generated action synchronises on completion, in interleaving, for each transition reaching the join pseudostate. At any moment this interleaving can be interrupted by a synchronisation on fire which restarts the join pseudostate. Next, the action checks if the guard of the outgoing transition is true. If it is, the orthogonal composite state that originates the incoming transitions is exited, each transition action is executed in interleaving, the action of the outgoing transition is executed and the target state is entered. If the guard is not true, the action waits for the next event fire (that indicates a potential change of state). Finally, the action recurses.

\footnote{constraint 2 on page 582 of \[Obj10b\] forbids triggers or guards in transitions reaching join pseudostates. While constraint 5 on the same page further restricts the outgoing transition not to have triggers, it does not forbid guards.}
The action generated by the next rule models a transition originating from a state reaching a fork pseudostate.

**Rule 5.57** \( \text{t\_fork}(f: \text{Pseudostate}; b: \text{Block})\): action declaration =

\[
\begin{align*}
\text{let int } &= \text{iota } t \text{ Transition } @ t.\text{target} = f \text{ in } \\
\text{trans\_name}(\text{int}) &= \mu X @ ( \\
&\text{name}(b) \text{ inevent!id(int)?e } \rightarrow \text{ Skip}; \\
&\text{t\_statecopy}(b,\text{name(int)}); ( \\
&\hspace{1em} \text{if } t.\text{triggers}(t.\text{trigger}) \text{ and } t.\text{guard} \text{ then enabled!id(int)!true } \rightarrow ( \\
&\hspace{2em} \text{fire } \rightarrow \text{ exit!id(int)!id(int.source) } \rightarrow \\
&\hspace{3em} \text{exited!id(int)!id(int.source) } \rightarrow \text{ Skip}; \\
&\hspace{3em} \text{t\_action}(t.\text{effect},b,\text{name(int)},int.\text{trigger}); ( \\
&\hspace{4em} \| \\
&\hspace{5em} \text{t\_action}(t.\text{effect},b,\text{name(int)},[]); \\
&\hspace{5em} \text{enter!id(int)!id(t.target) } \rightarrow \\
&\hspace{6em} \text{entered!id(int)!id(t.target) } \rightarrow \text{ Skip}) \\
&\hspace{4em} | \text{in set Transition } @ t.\text{source} = j) \\
&\hspace{3em} ) ) \text{ fired } \rightarrow \text{ Skip} \\
&\hspace{1em} |) \\
&\hspace{1em} \text{cancel?x!m!}(\text{mk\_token(name(int))) } \rightarrow \text{ fire } \rightarrow \text{ fired } \rightarrow \text{ Skip} \\
&\hspace{1em} |) \\
&\hspace{1em} \text{else enabled!id(int)!false } \rightarrow \text{ fire } \rightarrow \text{ fired } \rightarrow \text{ Skip} \\
&\hspace{1em} |)
\end{align*}
\]

This action initially offers a choice of receiving an event, or synchronising on fire. In the latter case, it then synchronises on fired. In the former case, the action reads the state components used in the guard, evaluates the trigger and the guard of the transitions that reaches the fork, and if they evaluate to true, signals that the transition is enabled, waits for fire and executes the behaviours associated with the transition. Next, in interleaving, it executes the behaviours associated with the transitions leaving the fork pseudostate and enters their target states. Once the interleaving terminates, it synchronises on fired. If the trigger and guard evaluate to false, the action synchronises first in fire and then on fired. Finally, in all cases, the action recurses.
5.2.9 Initial pseudostates

Initial pseudostates and the transitions that follow are not modelled as a separate action in our approach. The action derived from these elements is part of the action that models the state or region that contains the initial pseudostate as shown in Rule 5.12.

The UML standard [Obj11] is not consistent about the use of guards on transitions leaving initial pseudostates. For instance, the standard claims that “outgoing transitions from the initial vertex may have a behaviour, but not a trigger or guard” [Obj11, p. 551]. In a different section, it claims that “[t]ransitions outgoing pseudostates may not have a trigger (except for those coming out of the initial pseudostate)” [Obj11, p. 582]. In this translation, we assume that transitions leaving initial pseudostates do not have triggers or guards, and for all transition flows starting from an initial pseudostate, for all junctions in the flow, the conjunction of guards of its transitions is true. This is necessary, otherwise an initial pseudostate may not enter a state.

The rules are similar to those for transitions, except that:

- Initial pseudostates do not generate separate actions;
- Transitions in flows starting in initial pseudostates do not synchronise on the channels fire, fired, enabled, cancel or name(b)_inevent.
- The triggers of transitions leaving a junction (in the flow) form a cover.

**Rule 5.58**
\[
\text{t_initial_pseudostate}(s: \text{State}, b: \text{Block}): \text{action} = \\
\text{let } t = \text{iota x: Transition @ x.source = s.initial in} \\
\text{t_initial_to_vertex}(t.target, [t.effect], \text{name}(s), b) \\
\text{provided} \\
\quad 1. s \text{ in set State } => s.\text{isComposite} = \text{true and } s.\text{isOrthogonal} = \text{false or } s \text{ in set Region}
\]

**Rule 5.59**
\[
\text{t_init_to_vertex}(t: \text{Vertex}, as: \text{seq of Action}, n: \text{Name}, b: \text{Block}): \text{action} = \\
\text{if } \text{is_State}(t) \\
\text{then } \text{t_init_to_state}(t, as, n, b) \\
\text{else } \text{t_init_to_junction}(t, as, n, b) \\
\text{end if} \\
\text{provided} \\
\quad 1. \text{is_State}(t) \text{ or (is_Pseudostate}(t) \text{ and } t.\text{kind} = \#\text{junction})
\]
Rule 5.60 \texttt{t\_init\_to\_state(t:Vertex, as: seq of Action; n: Name; b: Block): action =}

\begin{align*}
\text{t\_actions(as,b,n,[])}; & \text{enter! [mk\_token(\texttt{n})]!id(t) } \rightarrow \\
& \text{entered! [mk\_token(\texttt{n})]!id(t) } \rightarrow \text{Skip}
\end{align*}

\texttt{provided}
\begin{enumerate}
\item \texttt{is\_State(t)}.
\end{enumerate}

Rule 5.61 \texttt{t\_init\_to\_junction(t:Vertex,as: seq of Action, n: Name, b: Block): action =}

\begin{align*}
\text{let ts = set2seq(y | y: Transition @ y.source = t) in} \\
\text{t\_statecopy(b,n);} \\
\text{if } \begin{small} & \begin{align*}
\text{t\_init\_from\_junction(x,as,n) } | & \text{x in set ts}
\end{align*}
\end{small} \text{fi}
\end{align*}

\texttt{provided}
\begin{enumerate}
\item \texttt{target in set \{x | x:Pseudostate @ x.kind = \#junction\}}.
\end{enumerate}

Rule 5.62 \texttt{t\_init\_from\_junction(t:Transition, as: seq of Action, n: Name): expression } \rightarrow \text{ action =}

\begin{align*}
\text{t.guard } \rightarrow & \text{ t\_init\_to\_vertex(t.target,as*[t.effect],n)}
\end{align*}

\texttt{provided}
\begin{enumerate}
\item \texttt{t.source in set \{x: Pseudostate | x.kind = \#junction \}}.
\item \texttt{t.target in set State union \{x: Pseudostate | x.kind = \#junction\}}
\end{enumerate}

5.2.10 Actions

The following rule translates an action without taking into consideration the context in which it is executed. For composite CML statements (e.g., sequential composition), the rule translates the subactions and combines them using the same combinator. For operation calls without return values, the rule generates an action that constructs a record of the type associated with the operation with the parameters of the call and sends it to the instance of the block (\texttt{id}) from the element that requested it (\texttt{[mk\_token(\texttt{n})]}), next it waits for an answer on the same channel. For operation calls with a return value, the action is similar, except that after receiving an answer, the component \texttt{_ret} of the record is assigned to the variable \texttt{v}.
A return statement is translated into a communication on the channel \( \text{op} \) with the source parameter being the identifier of the block treating the request, and target parameter the identifier of the block that requested the operation. The final parameter of the communication is an instance of the record type that models the output of the operation that is the trigger of the transition. The record is build using the variable \( v \) for the returned value. Finally, if the action is none of the above, it stays as it is, since it already is a primitive CML statement, like an assignment, for instance.

**Rule 5.63**

\[
\text{t_simple_action}(a: \text{Action}, b: \text{Block}, n: \text{Name}, trs: \text{seq of Trigger}): \text{action} =
\]

\[
\begin{align*}
\text{if } a &= ? \text{ as then } [\text{?} \{ \text{t\_action}(a,b,n,\text{trs}) \mid a \text{ in set as}\}]
\end{align*}
\]

\[
\begin{align*}
\text{elseif } a &= (\text{Op}(ps)) \text{ then }
& \text{name}(b).\text{op}.[\text{mk\_token}(n)].\text{id}.
& \text{mk\_name}(b)\_\text{types}\_\text{name}(\text{Op})\_\text{I}(ps) \rightarrow
& \text{name}(b).\text{op}.[\text{mk\_token}(n)].
& \text{id}\_x:(\text{is\_name}(b)\_\text{types}\_\text{name}(\text{Op})\_\text{I}(x)) \rightarrow \text{Skip}
\end{align*}
\]

\[
\begin{align*}
\text{elseif } a &= (v := \text{Op}(ps)) \text{ then }
& \text{name}(b).\text{op}.[\text{mk\_token}(n)].\text{id}.
& \text{mk\_name}(b)\_\text{types}\_\text{name}(\text{Op})\_\text{I}(ps) \rightarrow
& \text{name}(b).\text{op}.[\text{mk\_token}(n)].
& \text{id}\_x:(\text{is\_name}(b)\_\text{types}\_\text{name}(\text{Op})\_\text{I}(x)) \rightarrow v := x\_\text{ret}
\end{align*}
\]

\[
\begin{align*}
\text{elseif } a &= (\text{return } v) \text{ and len}(\text{trs}) = 1 \text{ and trs}(1) \text{ is Operation then }
& \text{name}(b)\_\text{op}.(e\_#1).\text{(e\_#2)}\_\text{mk\_name}(b)\_\text{name}(\text{trs}(1))\_\text{types}\_\text{O}(v) \rightarrow
& \text{Skip}
\end{align*}
\]

\[
\text{else } a
\]

where

1. \( ? \) stands for any statement constructor of CML (e.g., ;).

When executed in a state or transition, due to the parallelism present in the model, an environment must be created from the state before the action is executed. After the action is executed, the state must be updated. The state update can undo state changes performed in parallel, and parallel actions do not interfere with each other. A more selective form of parallelism of actions can be defined by modifying the action that updates the state.
Rule 5.64  \( t\_action(a: \text{Action}, b: \text{Block}, n: \text{Name}, \text{trs: seq of Trigger}): \text{action} = \)
\[
\begin{align*}
\text{t\_statecopy}(b,n); \\
\text{rename\_parameters}(\text{t\_simple\_action}(a,b,n,\text{trs})); \\
\text{t\_stateupdate}(b,n)
\end{align*}
\]

The action that updates the state is defined by the following translation rule. The rule takes a set of attributes, a block and the name of the element that executes the action (e.g., state); it produces an interleaving that writes each local variables generated by the action produced by Rule 5.40 to the state through the appropriate channel.

Rule 5.65  \( t\_stateupdate(\text{as: set of Attributes; b: Block; n: Name}): \text{action} = \)
\[
\begin{align*}
\text{|||} (\text{name}(b).\text{set\_name}(a).[\text{mk\_token}(n)].\text{id!}\text{name}(a) \rightarrow \text{Skip}) \mid a \text{ in as}
\end{align*}
\]

Finally, the following rule is similar to Rule 5.64 but applies to a sequence of actions. It created a local copy of the state (and copies the state to it), and translates each action of the sequence of actions using Rule 5.64. Finally, it updates the state with the value contained in the local copy of the state.

Rule 5.66  \( t\_actions(\text{as: seq of Action,b: Block, n: Name, trs: seq of Trigger}): \text{action} = \)
\[
\begin{align*}
\text{t\_statecopy}(b,n); \\
\text{for } a \text{ in as do} \\
\quad \text{rename\_parameters}(\text{t\_simple\_action}(a,b,n,\text{trs})); \\
\text{end for} \\
\text{t\_stateupdate}(b,n)
\end{align*}
\]

It simply applies the Rule 5.64 to each action in the sequence, and composes the resulting actions in sequence. The result of applying these rules to the action of the transition between the states empty and mid is presented below.

```plaintext
(dcl b: seq of Item @ (Buffer.get_b![\text{mk\_token(mid)}].id?x \rightarrow b:=x); b:=b^[e.#3.x]; Buffer.set_b![\text{mk\_token(mid)}].id!b \rightarrow \text{Skip})
```
In this action, a local copy of the state of the block Buffer is declared and the values stored in the block are copied to the local variables. Next, the assignment is executed with the parameter variable x replaced by the component of the record that models the operation call and is stored in the third position of the tuple e (this replacement is achieved through the rule rename_parameters). Finally, the local state is used to update the block’s state.

5.3 Related work

UML state machine diagrams have been extensively formalised. However, in most cases, key features, such as join, fork and history pseudostates, interlevel transitions and completion events are not covered. Moreover, in most cases the formalisation relies on certain restrictions to produce simpler models, making extensions of the approach difficult, if not impossible, without comprehensive re-working of the formalisation. For instance, an assumption that there are no interlevel transitions yields a very simple model and translation rules, which cannot be directly extended to account for interlevel transitions.

[Wha10] proposes a parametrised structural operational semantics that can be instantiated for UML state machine diagrams. A structural operational semantics describes the result of executing a program (or chart) in terms of the execution of its parts. [LMM99] proposes a translation strategy between UML state machine diagrams and extended hierarchical automata, which are automata extended with hierarchy (sequential and parallel composition) and information that allows particular substates to be entered. Additionally, [LMM99] defines an operational semantics for such automata. Similarly, [LP99] translates UML state machine diagrams into PROMELA, the input language for the model checker SPIN, to support the analysis of the diagrams. All these works focus on verification of properties of UML state machine diagrams in isolation. Moreover, none of these approaches directly support refinement, thus, support to this technique requires the development of new theories and techniques.

[NB03] translates UML state machine diagrams to CSP processes by associating states to CSP processes, and UML events to CSP events. The execution of multiple transitions is modelled by external and internal choice depending on the nature of the transitions (explicitly or implicitly triggered). Due to the limited subset of state machine diagrams that is covered in this work, the benefits of reusing these results would be small when compared with the effort necessary to extend the approach and integrate in a semantics of SysML that takes different diagrams into account.
Table 5.1: Coverage comparison

[RS05] proposes a mapping between UML-RT active classes (that include state machine diagrams) to Circus. This work extends that of [NB03] by covering untreated aspects such as AND states. Furthermore, [RS05] discusses the use of the Circus refinement calculus to support model transformation on the UML level. As in the case of [NB03], this approach proved difficult to extend for the features that were left untreated.

Table 5.1 presents a comparison (partially based on [CD05]) of the coverage of our formalisation with some of available literature. The top row contains the features that we cover: simple states (SS), composite states (CS), final states (FS), initial pseudostate (IP), regions (Re), completion transitions (cT), interlevel transitions (iT), events (E), deferred events (dE), join pseudostate (j), fork pseudostate (f) and action language (a). The ✓ indicates that the feature is covered by the work on the left most column, and × indicates it is not.

To the best of our knowledge [MN04] is the only work that covers refinement, none of them provide an integration with the semantics of other diagrams.
Chapter 6

Sequence diagrams

The translation rules that cater for the link between SysML sequence diagram and CML are presented and illustrated with the help of an example. Such translation is divided in the basic features of SDs and more complex constructs presented further. To illustrate the basic features we use the same example of the Producer-Consumer where one possible scenario is depicted in a Sequence Diagram using only its basic elements.

This chapter is organised as follows. Section 6.1 describes the example we use to illustrate our translation rules, which are introduced in Section 6.2. Related work is presented in Section 6.3.

6.1 Examples

The example below depicts a scenario having three entities and their respective lifelines: one producer, one buffer and one consumer. The behaviour illustrated is a very simple possibility of execution where the producer adds one item to the buffer, then the consumer removes such item from the buffer. The add operation is done asynchronously whilst the remove operation is synchronous and the respective reply message consists of the element retrieved. This can be seen in Figure 6.1.

Another way of represent this scenario is using InteractionUse. This is illustrated by figures 6.2 and 6.3. They show the same behaviour of the previous sequence diagram, however, we define a specific diagram to illustrate the removal of a item. This diagram is used inside the other as an InteractionUse. The first diagram displayed in Figure 6.2 shows an asynchronous message arriving from outside to
the Buffer, and then the Consumer performs a synchronous operation. Figure 6.3 illustrates the use of the previous one by the InteractionUse element.

Based on these examples we describe the translation functions for the basic features of a sequence diagram plus the mechanisms to translate InteractionUse elements.
6.2 Formal semantics

Each translation rule can be regarded as particular pattern matching case of a recursive function that is applied top-down. The translation starts from the entire sequence diagram and recursively goes down to the level of a message transmission, which is the base of the recursion. The semantics of a sequence diagram is a CML specification that denotes a set of traces that records all messages of the diagram. A message in a trace is a record that contains all information about that message. For example, we store the id of the sender, the id of the receiver, the id of the message, the values of the parameters etc.

We assume that all elements translated have unique identifiers, here we use the attribute "index", so we can differentiate elements of the same type. For example, two or more BREAK or PAR combined fragments. We also suppose that we have some auxiliary functions available, for example, \texttt{lf\_channels(if: seq of InteractionFragments, lf: Lifeline)} from Rule 6.4 which should return the channels used inside the translation of a lifeline. These function are explained along the text as they are needed.

6.2.1 Root Sequence Diagram

We start with the root function for sequence diagrams with no InteractionUse (diagrams used inside another). A more complex rule is detailed in Section 6.2.6 to cater for the use of InteractionUse.
Rule 6.1 \( t\_\text{sequencediagram}(sd: \text{Interaction}): \text{seq of program paragraph} = \)

channels

\[
\text{join, break, interrupt: } \text{nat;}
\]

strict: \( \text{nat.nat} \) beginCR, endCR: \( \text{nat;} \)

startRef, endRef: \( \text{nat;} \)

endInteraction: ID;

invalid\_trace, block: ID;

if sd.hasInteractionUse then

\( t\_\text{complex\_sd}(sd) \)

else

\( t\_\text{simple\_sd}(sd, \text{false}) \)

Rule 6.1 defines channel declarations for sequence diagrams. These channels are for internal use of the sequence diagrams model and depending on the existence or not of InteractionUse elements, two different functions may be invoked. First, we detail the construction of the model for simple sequence diagrams according to the second function \( t\_\text{simple\_sd}() \).
Rule 6.2 \( t\_simple\_sd(sd: \text{Interaction}, \text{internal}: \text{Boolean}): \text{seq of program paragraph} = \)

\[
\begin{align*}
\text{if internal then} & \quad \text{process } \text{sd}_\text{internal}_\text{name}(sd) = \text{sd}_\text{id}: \text{ID}; \\
\text{else} & \quad \text{process } \text{sd}_\text{name}(sd) = \text{sd}_\text{id}: \text{ID}; \\
\text{end if} \\
\text{for } \text{lf} \text{ in seq } \text{sd}.\text{Lifelines do} & \quad \text{t}_\text{lifeline}_\text{name(lf)}\_\text{id}: \text{ID}; \\
\text{end for} \\
\text{for } \text{gate} \text{ in seq } \text{sd}.\text{formalGates do} & \quad \text{name(gate)}\_\text{id}: \text{ID}; \\
\text{end for} \\
\text{let } \text{paramList} = \text{[]} \text{ in} & \quad \text{for } \text{m} \text{ in seq } \text{sd}.\text{Messages do} \\
\text{for } \text{p} \text{ in seq } \text{m}.\text{Arguments do} & \quad \text{if } \text{p}.\text{isInteractionParameter()} \text{ and not exists(p.name, paramList) then} \\
& \quad \text{p.name} : \text{t_types(p.type)} ; \\
& \quad \text{paramList} = \text{paramList union p.name} \quad \text{end if} \\
\text{end for} \\
\text{end for} \\
\text{end} \\
\end{align*}
\]

Rule 6.2 defines the process for the diagram with a list of parameters, which represent the ID for the sequence diagram, IDs for the lifelines, IDs for gates, arguments used in the messages that are defined as parameters of the Interaction, respectively. For the latter, we use a let expression to define a set \text{paramList} that keeps the parameters already defined and avoid duplicated parameter definitions. We check if the parameter has not been created yet through the use of function \text{exists()}, which receives a element and a set and it checks if the element exists in the given set. The second argument of the translation function determines if the CML process to be created is for an internal sequence diagram, which is used when defining diagrams with InteractionUse elements (more details in Section 6.2.6), or a usual one. Such verification only alters the name of the CML process that is being created. Finally, we have the channel sets and actions declarations that will be explained further by Rules 6.4 and 6.5. The lifelines name is acquired using an auxiliary function (\text{t_lifeline}_\text{name(lf)}) that will be explained.
next by Rule 6.3.

**Rule 6.3** \( t_{\text{lifeline} \_\text{name}}(l{\text{f}}: \text{Lifeline}) : \text{action} = \)

\[
\begin{align*}
\text{if } \text{lf.represent\_selector} \text{ == null then} \\
\quad \text{name(lf.represent\_selector)} \\
\text{else} \\
\quad \text{name(lf.represent\_selector)}_{\text{lf.represent\_name(lf.represent)}} \\
\text{end if}
\end{align*}
\]

If there is no instance name it returns the name of the block, otherwise, it returns the name of the instance appended by an underscore \(_\) followed by the name of the block. This rule is used inside other rules along this document.

Rule 6.4 states how the channel sets for a sequence diagram should be defined. This rule is being called from the Rule 6.2. It defines the set of internal control channels that are hidden. Then, for each lifeline it defines channel sets by calling the function \( \text{lf\_channels(lf.InteractionFragments, lf)} \). Each of these channel sets represent an alphabet of synchronisation of the CML action that models the corresponding lifeline. The function \( \text{lf\_channels()} \) simply defines a channel set that the action that models the lifeline uses. This means that message event channels and also control channels may appear. Control channels are introduced for some CombinedFragments like BREAK (explained in Rule 6.19). A message channel stores information about the message itself, its sender ID and its receiver ID (see rules from Section 6.2.3). A control channel stores information about its communicated data. The channel set \( \text{Events} \) represents the set of all events used by this sequence diagram. This channel set is used in Rule 7.30.

**Rule 6.4** \( t_{\text{sd\_chansets}}(sd: \text{Interaction}) : \text{seq of program \_paragraph} = \)

\[
\begin{align*}
\text{channelsets} \\
\text{Hidden} = \{ \text{join, break, invalid\_trace, endInteraction,} \\
\quad \text{strict, beginCR, endCR} \} \\
\text{for } \text{lf in seq sd.Lifelines do} \\
\quad \text{cs}_{t_{\text{lifeline\_name(lf)}}} = \text{lf\_channels(lf.InteractionFragments, lf)} \\
\text{end for} \\
\text{Events} = \text{union} \\
\text{for } \text{lf in seq sd.Lifelines do} \\
\quad \text{cs}_{t_{\text{lifeline\_name(lf)}}} \\
\text{end for}
\end{align*}
\]

137
The application of Rules 6.2, 6.3 and 6.4 to the example results in the following CML extract:

```cml
channels
    join, break, interrupt: nat;
    strict: nat.nat;
    beginCR, endCR: nat;
    startRef, endRef: nat;
    endInteraction: ID;
    invalid_trace, block: ID;
process sd_add_rem = sd_id: ID; Consumer_id: ID; Buffer_id: ID;
    Producer_id: ID; item: Item; @ begin
chansets
    Hidden = {join, break, invalid_trace, endInteraction,
        strict, interrupt, beginCR, endCR}
    cs_Consumer = ...
    cs_Buffer = ...
    cs_Producer = ...
    Events = cs_Consumer union cs_Buffer union cs_Producer
actions
    ...
end
```

First, we have the channel section having all control channels needed when defining sequence diagrams in CML. Next, we have the CML process signature having by name `sd_` appended by the name of the diagram (`add_rem`), followed by the process parameters, the ids for the diagram, Consumer lifeline, Buffer lifeline, Producer lifeline, and the parameter of the Interaction (`item`), respectively. Next, we have the chanset definitions, first the `Hidden` chanset followed by the chanset of each lifeline. The last chanset is the one that represents all possible events of the diagram. Finally, we have the action definitions and the end of the CML process.

Rule 6.5 details the action section of the CML process, which contains actions for the lifelines and auxiliary actions for specific combined fragments. At the end of this rule we show the definition of the main action declaration.
D22.4 - SysML–CML (Public)

**Rule 6.5**
\[
\text{t_sd_actions(sd: Interaction): seq of program paragraph} =
\]

```plaintext
actions

for lf in seq sd.Lifelines do
  t_lifeline_name(lf) = t_lf_interaction_fragments(lf.InteractionFragments, lf)
  if lf.hasLoopFragments() then
    t_loop_actions(lf.getLoopCombinedFragments, lf)
  end if
end for

if sd.hasCriticalFragments() then
  t_critical_actions(sd.getCriticalCombinedFragments, sd)
end if

if sd.hasBreakCombinedFragment() then
  BREAK = break?i -> interrupt!i -> BREAK
  [] endInteraction.sd_id -> Skip
end if

end t_main_action(sd)
```

We translate each lifeline to CML by calling `t_lf_interaction_fragments()`. Then, the conditional statement checks if there are any LOOP fragments inside the lifeline in order to create auxiliary actions needed as a point of recursion for loops (Rule 6.17). The second conditional statement adds auxiliary actions for the treatment of the CRITICAL CombinedFragment in case there is one or more of them (Rule 6.20). The third conditional statement adds an auxiliary action in case any BREAK CombinedFragment exists in order to halt the flow of a lifeline. A BREAK has a similar behaviour to an exception mechanism. If the BREAK condition is true, then some actions are executed followed by the interruption of the flow of the lifeline. The auxiliary action for the BREAK shown in Rule 6.5 catches a BREAK interruption. A more detailed explanation on CombinedFragments is given in Section 6.2.4.

The result of the application of Rule 6.5 to our example should give the following CML extract:

```plaintext
actions
  lf_Producer = ...
```

139
In this extract we have the definition of each lifeline, Producer, Buffer and Consumer, and any auxiliary actions needed. After the @ character we have the definition of the main action. The translation of these element is presented next.

### 6.2.2 Lifeline

Rule 6.6 shows how the InteractionFragment elements from a lifeline are translated. The first for iterates over all fragments and calls the appropriate translation function according to the type of the fragment. These InteractionFragment elements are composed sequentially using the operator ;. The last for command adds interruption events in case any of these fragments are the BREAK CombinedFragments. Such events will halt the flow of fragments leading them to Skip.

**Rule 6.6 t_lf_interaction_fragments(ifs: seq of InteractionFragment, lf: Lifeline): action =**

```plaintext
( ;
  for intf in seq ifs do
    switch(intf.Type)
    case MessageType: t_message(intf)
    case CombinedFragmentType: t_combined_fragment(intf, lf)
    case StateInvariantType: t_state_invariant(intf, lf)
    case InteractionUseType: t_interaction_use(intf)
    end switch
  end for
)

for intf in seq ifs do
  if intf.Type == CombinedFragmentType and intf.cfType == BREAK.Type then
    interrupt. intf.index -> Skip
  end if
end for
```

140
6.2.3 Messages

The next rule details the translation of message communications. Depending on the type of the message, specific rules are invoked. The second and third arguments of these message translation functions are lifelines. However, if this message comes from or goes to a gate, then instead of a lifeline, the gate is passed as argument.

**Rule 6.7**

\[
\text{t_message(mos: MessageOccurrenceSpecification): action =}
\]

\[
\text{switch(mos.Message.messageSort)}
\]

\[
\begin{align*}
\text{case synchCall:} & \quad \text{t_synch_call(mos, mos.Message.sendEvent.Lifeline, mos.Message.receiveEvent.Lifeline)} \\
\text{case asynchCall:} & \quad \text{t_asynch_call(mos, mos.Message.sendEvent.Lifeline, mos.Message.receiveEvent.Lifeline)} \\
\text{case reply:} & \quad \text{t_reply_call(mos, mos.Message.sendEvent.Lifeline, mos.Message.receiveEvent.Lifeline)}
\end{align*}
\]

\[\text{end switch}\]

Rule 6.8 translates the occurrence of sending or receiving a synchronous message. First it verifies if it is a sending or receiving event. In case of a sending event, the action defined collects the value of the attributes from the sending lifeline that are used as arguments to the message. Here we reuse Rule 5.40, which defines an action where local copies of these attributes are created and used later when iterating the arguments. However, a simple version of this action may be formulated where we only collect the values from the attributes that appear in the referred message.

Another comment on arguments is that we only consider attributes from blocks or parameters of the Interaction (constants, wildcards, and so on). Both are matched according to their names that should be the same as \text{name(arg)}. However, some may require expressions as arguments, which are not existent in the SysML/UML specification [Obj11] and, thus, not considered in this translation function.

After the copy of state from attributes, the CML event corresponding to the message is built according to the message constructs. Note that the sending occurrence has two events: one for synchronising the sending and other for the receiving. The interpretation for synchronous call is that the sender stays suspended waiting for the response. In case the occurrence is a receiving event, only one CML event is built (because it may perform other tasks to fulfill the receiving call).
Rule 6.8 \texttt{t_synch\_call}(mos: MessageOccurrenceSpecification, sender, receiver; Lifeline):

\begin{verbatim}
if mos.isSendEvent() then
    t_statecopy(receiver.represents, mos.Message)
    name(receiver.represents).op.mos.Message.id.
        t_lifeline_name(sender)_id.
        t_lifeline_name(receiver)_id!mk_name(receiver.represents)
        -> name(receiver.represents).op.mos.Message.id.
            t_lifeline_name(sender)_id.
            t_lifeline_name(receiver)_id?out: name(receiver.represents)_types’
            mos.Message.signature_O) -> Skip
else
    name(receiver.represents).op.mos.Message.id.
        t_lifeline_name(sender)_id.
        t_lifeline_name(receiver)_id?oper: name(receiver.represents)_types’
        mos.Message.signature_I) -> Skip
end if
\end{verbatim}

In our example, the Consumer performs a synchronous call invoking the \textit{rem} operation. The following extract details the translation of the Consumer lifeline including this message:

\begin{verbatim}
lf_Consumer = ((Buffer.op.2.Consumer_id.Buffer_id!
    mk_Buffer_types’rem_I() ->
    Buffer.op.2.Consumer_id.Buffer_id?
    out: Buffer_types’rem_O) -> Skip))
\end{verbatim}

Here we have an example of synchronous message \textit{rem} from the perspective of the sender (Consumer). The first channel details the sending message, and it uses an operation of the Buffer block (Buffer.op). Next, we have the index of the message, followed by the ids of the sender (Consumer) and the receiver (Buffer). The last part of the event details the signature of the operation, which is a type of the Buffer block (operations are treated as types of blocks as detailed in Chapter 4), and signature \textit{rem}. The appended \_I means that it is a sending event with any input parameters needed. As the \textit{rem} does not require any input parameter, it is empty.

The second event follows the same idea of the first, however, it is waiting for a response of the Buffer block (?out...) with may come with output parameters (we treat return values and output parameters the same way). That is why the signature of the operation is followed by \_O. This sequencing of two events blocks the sender to proceed to the next occurrence in the lifeline until it receives
a response of the receiver.

Rule 6.9 describes how to translate asynchronous messages. It is very similar to the previous rule. However, the sender does not wait for a response, hence, it may proceed to the next occurrence after the sending event. The receiver waits for the same event and treats the communication according to the asynchronous mechanism provided by the event pool, which is specified by the model of other diagrams (State Machine or Activity). As the treatment of asynchronous messages is identical to that of signals, we consider the channels that deal with signals in the blocks to represent asynchronous communication. Thus, we only need one event to represent such messages instead of $op$ for operation, we have $sig$ for signal, and at the end of the type declaration instead of $I$ or $O$, we have $S$.

**Rule 6.9**

```
rule t_asynch_call(mos: MessageOccurrenceSpecification, sender, receiver: Lifeline): action =
if mos.isSendEvent() then
    t_statecopy(receiver.represents, mos.Message)
    name(receiver.represents).sig.mos.Message.id. t_lifetime_name(sender)_id.
    t_lifetime_name(receiver)_id!mk_name(receiver.represents)
    _types’mos.Message.signature_S( mos.Message.Arguments) ) -> Skip
else
    name(receiver.represents).sig.mos.Message.id. t_lifetime_name(sender)_id.
    t_lifetime_name(receiver)_id?signal: name(receiver.represents)_types’
    mos.Message.signature_S -> Skip
end if
```

To illustrate an asynchronous message translation, we show an extract of the example of the Producer lifeline:

```
lf_Producer = ((Buffer.sig.1.Producer_id.Buffer_id!mk_Buffer_types’add_S(item) -> Skip))
```

The sending of an asynchronous message $add$ is represented by a single event which follows a similar format to that for a synchronous message, where the differences are: instead of $op$ we have $sig$, as we treat asynchronous messages the same manner of signals, and the end of the signature is followed by a $S$, for "signal". As it performs such event it may proceed to the next occurrence, different from synchronous messages that wait for a reply.

Rule 6.10 shows the rule for reply calls. This is a simpler rule as we just need
to translate the sending of the reply as the receiving is guaranteed by Rule 6.8. Both events are waiting for something (?out). This happens because the model of the sequence diagram does not perform the operation related to the call; this should be done by the model of the other diagrams. Thus, the process that models the sequence diagram must synchronise with the processes that model these other diagrams to obtain the output.

**Rule 6.10**

\[
\text{t_reply_call(mos: MessageOccurrenceSpecification, sender, receiver: Lifeline):}
\]

\[
\text{action =}
\]

\[
\begin{cases}
\text{if mos.isSendEvent() then} \\
\quad \text{name(receiver.represents).op.mos.Message.id.}_t\_\text{lifeline_name(sender)}\_id. \\
\quad \text{t_lifeline_name(receiver)}\_id?\text{out: name(receiver.represents)} \\
\quad \_\text{types'}mos.Message.signature}_O \rightarrow \text{Skip}
\end{cases}
\]

end if

To display an example of all kinds of messages, the next extract shows the Buffer lifeline translation (note that each occurrence is composed with the sequential composition operator [;]):

\[
\text{lf_Buffer = ((Buffer.sig.1.Producer_id.Buffer_id?signal: Buffer_types'}\text{add_S} \rightarrow \text{Skip});}
\]

\[
\quad \text{(Buffer.op.2.Consumer_id.Buffer_id?oper: Buffer_types'}\text{rem_I} \rightarrow \text{Skip});}
\]

\[
\quad \text{(Buffer.op.2.Consumer_id.Buffer_id?out: Buffer_types'}\text{rem_O} \rightarrow \text{Skip}})
\]

To complete the description of the translation for sequence diagrams, we show the Rule 6.11 and auxiliary actions in parallel. Rule 6.11 describes the main action of the CML process that models a sequence diagram. It reuses the function from Rule 4.21 to create an alphabetised parallelism among the lifelines, where a series of other parallelisms are added. If there is any BREAK CombinedFragment, then the result of the application of the respective auxiliary action, which is defined in Rule 6.5, is added in parallel. This is needed due to the nature of BREAK fragments, which depends on the context where the fragment is inserted. Finally, in case of the existence of any CRITICAL fragment, the related action, which is shown in Rule 6.20, is also allocated in parallel. The control channels, that is, those in the set Hidden, are hidden.
Rule 6.11 \( \text{t\_main\_action}(sd: \text{Interaction}); \text{action} = \)

\[
\text{@ (define_alphabetised_parallel}((\text{t\_lifeline\_name}(s). \text{cs\_t\_lifeline\_name}(s)) \text{s in set sd.Lifelines}); \text{endInteraction.sd_id} \rightarrow \text{Skip}
\]

\[
\text{if sd.existsBreakCombinedFragment}() \text{then}
\]

\[
\left\{\left\{\text{break, interrupt, block, endInteraction}\right\}\right\} \text{BREAK}
\]

\[
\text{end if}
\]

\[
\text{if sd.existsCriticalFragments}() \text{then}
\]

\[
\left\{\text{Events}\right\} \text{CRITICAL}
\]

\[
\text{end if}
\]

\[
\text{Hidden}
\]

Finally, with the presented rules we are able to show in the next extract the complete CML process related to our example.

channels

\[
\text{join, break, interrupt: nat; strict: nat.nat; beginCR, endCR: nat; startRef, endRef: nat; endInteraction: ID; invalid\_trace, block: ID; process sd_add\_rem = sd\_id: ID; Consumer\_id: ID; Buffer\_id: ID; Producer\_id: ID; item: Item; @ begin chansets Hidden = \{join, break, invalid\_trace, endInteraction, strict, interrupt, beginCR, endCR\}
\]

\[
\text{cs\_Consumer = ...}
\]

\[
\text{cs\_Buffer = ...}
\]

\[
\text{cs\_Producer = ...}
\]

\[
\text{Events = cs\_Consumer union cs\_Buffer union cs\_Producer}
\]

actions

\[
\text{lf\_Consumer} = ((\text{Buffer\_op.2.Consumer\_id.Buffer\_id! mk\_Buffer\_types’rem\_I() -> Buffer\_op.2.Consumer\_id.Buffer\_id? out: Buffer\_types’rem\_O) \rightarrow \text{Skip})}
\]

\[
\text{lf\_Producer} = ((\text{Buffer\_sig.1.Producer\_id.Buffer\_id! mk\_Buffer\_types’add\_S(item) -> Skip})
\]

\[
\text{lf\_Buffer} = ((\text{Buffer\_sig.1.Producer\_id.Buffer\_id? signal: Buffer\_types’add\_S -> Skip})
\]
This last extract displays the conjunction of the previous ones. First, we have channel and chanset definitions, and the process signature. Next, we have the definition of CML actions for each of the lifelines. Finally, the main action is organized according to the arrangement of all lifelines in a composed alphabetised parallelism where the synchronisation sets are the chansets previously defined.

Next we present some other elements that increase the expressibility of sequence diagrams.

6.2.4 Combined Fragments

Here we show some rules for CombinedFragments. They work as operators that may impose other orders and meanings to the basic flow of the time axis of lifelines.

Rule 6.12 shows the basic rule when the element to be translated is a CombinedFragment. It calls the appropriate function according to the type of the fragment.
Rule 6.12 \( t_{\text{combined\_fragment}}(\text{cf}: \text{CombinedFragment}, \text{lf}: \text{Lifeline})\): action =

\[
\text{switch(}\text{cf.cfType})
\begin{align*}
\text{case PAR.Type: } & t_{\text{par\_combined\_fragment}}(\text{cf}, \text{lf}) \\
\text{case STRICT.Type: } & t_{\text{strict\_combined\_fragment}}(\text{cf}, \text{lf}) \\
\text{case ALT.Type: } & t_{\text{alt\_combined\_fragment}}(\text{cf}, \text{lf}) \\
\text{case OPT.Type: } & t_{\text{opt\_combined\_fragment}}(\text{cf}, \text{lf}) \\
\text{case BREAK.Type: } & t_{\text{break\_combined\_fragment}}(\text{cf}, \text{lf}) \\
\text{case LOOP.Type: } & t_{\text{loop\_combined\_fragment}}(\text{cf}, \text{lf}) \\
\text{case CRITICAL.Type: } & t_{\text{critical\_combined\_fragment}}(\text{cf}, \text{lf})
\end{align*}
\text{end switch}
\]

PAR CombinedFragment

The PAR CombinedFragment is a parallel operator that takes several operands and runs them in interleaving (\( ||| \)). Rule 6.13 shows the translation of the PAR CombinedFragment. The translation of the operands is specified by Rule 6.6, which translates a sequence of InteractionFragments. As we are translating a lifeline, we consider that the function \( \text{op.InteractionFragmentsFromLifeline()} \) only returns the sequence of InteractionFragments of the lifeline passed as argument. We also consider that all lifelines must synchronise at the end of the PAR CombinedFragment, that is why we need the control channel \( \text{join} \), which is sequentially composed with the parallelism. As all lifelines synchronise in this event, they will not follow to the next step before the others end their execution inside the fragment.

Rule 6.13 \( t_{\text{par\_combined\_fragment}}(\text{par}: \text{CombinedFragment}, \text{lf}: \text{Lifeline})\): action =

\[
( ||| \{ t_{\text{lf\_interaction\_fragments}}(\text{op}\text{.InteractionFragmentsFromLifeline(lf), lf}) | \text{op in seq par.Operands}) \} ; \text{join}.\text{par.index} \rightarrow \text{Skip}
\]

STRICT CombinedFragment

Rule 6.14 defines the translation for the STRICT CombinedFragment. The semantics states that the operands of this fragment must be executed in a top-down manner. Thus, the top level operand is executed first, then the second is performed and so on. Thus, we just need to synchronise the end of each operand to allow...
the beginning of the next. The control event is $strict.\$i.\$j$, where $i$ and $j$ are, respectively, a unique identifier for the current strict sequencing fragment and an index for each of the operands of this fragment. As the lifelines involved in each operand must synchronize before going to the next, we assure a strict order of execution.

Rule 6.14 $\text{t\_strict\_combined\_fragment(\text{strict: CombinedFragment, \text{lf: Lifeline}): action =}$

\[
\begin{align*}
{\text{t\_lf\_interaction\_fragments(op.InteractionFragmentsFromLifeline(lf), lf);}} \\
{\text{strict.\text{strict\_index}.\text{op\_index} \rightarrow \text{skip} \mid \text{op in seq strict.Operands}}} 
\end{align*}
\]

ALT CombinedFragment

Rule 6.15 details the alternative CombinedFragment (ALT) translation. The operands may have guards or not. If they do, we have to evaluate the guards. As they may use attributes from the lifelines, we have to create local definitions of these values to evaluate the guards consistently across all lifelines. This is necessary as the model of the lifelines run in parallel.

We consider that the function $\text{t\_statecopy\_Constraint(\text{operand\_Constraint, \text{alt}})}$, which is a variation of the one defined in Rule 5.40, should traverse the guard (also called a constraint in SysML) and create local definitions for variables that are attributes from lifelines. The id used comes from the CombinedFragment passed as the second argument. The next step is to construct the evaluation of the alternatives. This is done through the use of a nested if-then-else expression together with the same function used in the previous rule to translate the contents of operands. On the other hand, when there are no guards, the choice on the operand must be performed non-deterministically. This is translated using the internal choice operator.
Rule 6.15  \( t_{\text{alt\_combined\_fragment}}(\text{alt}: \text{CombinedFragment}, \text{lf}: \text{Lifeline}): \text{action} = \)

\[
\text{if } \text{alt.OperandsHaveGuards} \text{ then }
\begin{align*}
\quad & \text{for operand in seq alt.Operands do} \\
\quad & t_{\text{statecopyConstraint}}(\text{operand.Constraint}, \text{alt}) \\
\quad & \text{end for}
\end{align*}
\]

\[
\text{else for operand in seq alt.Operands do} \\
\quad \text{if operand.InteractionConstraint} \neq \text{"else" then} \\
\quad \quad \text{if operand.InteractionConstraint.specification then} \\
\quad \quad \quad t_{\text{lf\_interaction\_fragments}}(\text{operand.InteractionFragmentsFromLifeline(lf)}, \text{lf}) \\
\quad \quad \text{else} \\
\quad \quad \quad t_{\text{lf\_interaction\_fragments}}(\text{operand.InteractionFragmentsFromLifeline(lf)}, \text{lf}) \\
\quad \text{end if} \\
\quad \text{end for}
\]

\[
\text{else} \\
\quad ( \{ t_{\text{lf\_interaction\_fragments}}(\text{op.InteractionFragmentsFromLifeline(lf)}, \text{lf}) \mid \text{op in seq alt.Operands} \})
\]

\end{align*}
\]

end if

OPT CombinedFragment

Rule 6.16 defines translation function for the option CombinedFragment (OPT). It is a simplification of the previous rule, as this fragment always has only one operand that may be executed according to the guard.

Rule 6.16  \( t_{\text{opt\_combined\_fragment}}(\text{opt}: \text{CombinedFragment}, \text{lf}: \text{Lifeline}): \text{action} = \)

\[
t_{\text{statecopyConstraint}}(\text{opt.Operand.Constraint}, \text{opt})
\]

\[
\text{if } \text{opt.Operand.InteractionConstraint.specification} \text{ then} \\
\quad t_{\text{lf\_interaction\_fragments}}(\text{opt.Operand.InteractionFragmentsFromLifeline(lf)}, \text{lf}) \\
\text{else Skip} \\
\]

\end{align*}
\]

LOOP CombinedFragment

Regarding the LOOP CombinedFragment two functions are related to the translation. The first appears in Rule 6.5 and it is responsible for the creation of auxiliary actions that allow the iteration mechanism using recursion. The definition of this function is presented in Rule 6.17. The other rule exists simply for
adding to the model of the lifeline the right loop auxiliary action; it is presented in Rule 6.18.

Rule 6.17 creates the auxiliary loop actions needed for recursion. SysML provides a loop that takes parameters as input and behaves like a “for” loop. Depending on the number of parameters for iteration, it can create three types of actions: with minimum and maximum number of iterations, only with maximum number of iterations, and with no parameters for iterations. Another aspect is the existence or not of InteractionConstraint, which changes the conditional evaluation of the loop.
Rule 6.17  \( t_{\text{loop\_actions}}(\text{loops: seq of CombinedFragment, \text{lf: Lifeline})}: \text{action} = \)

for loop in seq loops do
  switch (loop.numberOfParameters)
  case 2:
    if (counter < min) or (counter >= min and counter <= max)
    end if
    end for
  end switch

Rule 6.18 is the invocation of the auxiliary actions defined in the previous rule. If the loop has an InteractionConstraint, the local copies of possible attributes of
blocks must be declared using the same function used previously \((t\textunderscore \text{statecopyConstraint})\). This is needed as the models of each lifeline run in parallel, so we have to evaluate the guard (constraint) consistently in all of them. Then, according to the number of parameters for iteration the appropriate action is added to the sequence of events from a lifeline. At the end we verify if the loop can be interrupted by a BREAK fragment adding the respective event to halt the loop.

**Rule 6.18**

\[
\text{t.loop\_combined\_fragment}(\text{loop: CombinedFragment, If: Lifeline}): \text{action} =
\]

```plaintext
if loop.hasInteractionConstraint() then
  \(t\textunderscore \text{statecopyConstraint}(\text{loop.InteractionConstraint, loop})\)
end if

switch (loop.numberOfParameters)
  case 2:
    \(\text{lf.Name\_LOOP_}\text{loop.index (1, loop.firstParameter, loop.secondParameter)}\)
  case 1:
    \(\text{lf.Name\_LOOP_}\text{loop.index (1, loop.firstParameter)}\)
  case 0:
    \(\text{lf.Name\_LOOP_}\text{loop.index}\)
end switch
for intf in seq loop.InteractionFragments do
  if intf.Type == CombinedFragmentType and intf.cfType == BREAK.Type then
    \(/\text{ interrupt. intf.index} \rightarrow \text{Skip}\)
  end if
end for
```

**BREAK CombinedFragment**

Rule [6.19] describes the function for the BREAK CombinedFragment. The action generated by this translation is simple, however it only works together with the actions translated content added by other functions from Rules [6.5], [6.6], [6.11] and [6.18] as these rules insert listeners that catch the interruption of a BREAK. The BREAK is a complex operator because it depends on the context where it is inserted. If the InteractionConstraint is evaluated to true, the action that models the scenario from the operand is performed instead of the action that models the remainder of the enclosing InteractionFragment, otherwise the opposite occurs. Hence, it only affects the InteractionFragment that encloses it. When the InteractionConstraint is true, the break event synchronises with the BREAK action from Rule [6.5], which fires the interrupt event. Such event halts the sequential flow in rules [6.6] and [6.18]
Rule 6.19 \texttt{t_break_combined_fragment(brk: CombinedFragment, lf: Lifeline): action =}
\begin{verbatim}
t_statecopyConstraint(brk.Operand.Constraint, brk)
if brk.Operand.InteractionConstraint.specification then
  t_lf_interaction_fragments(brk.Operand.InteractionFragmentsFromLifeline(lf), lf)
  ; break.brk.index -> block -> \texttt{Skip}
else \texttt{Skip}
\end{verbatim}

\textbf{CRITICAL CombinedFragment}

Rules \cite{6.21} and \cite{6.20} define the translations for critical regions. The semantics states that whilst inside a critical region no other Occurrence may be interleaved with the ones inside the fragment, even if the context says so, for example, inside a PAR CombinedFragment. The content of the critical region must run atomically. This is a complex operator to represent because it also depends on the context it is inserted. Few formalisation approaches in fact define it and mostly in a unclear way \cite{Sto04}.

To restrict the execution just to the critical region alone we need to create an controller action $C_{R}$, where $i$ is the unique index for the critical region. This is performed by Rule \cite{6.20}, which is invoked by Rule \cite{6.5}

Rule 6.20 \texttt{t_critical_actions(crts: seq of CombinedFragment, sd: Interaction): action =}
\begin{verbatim}
for crt in seq crts do
  CR.crt.index = RUN(ev(sd) - ev(crt)) \/
  beginCR.crt.index -> RUN(ev(crt)) \/
  endCR.crt.index -> CR.crt.index
end for
CRITICAL = \texttt{Skip}
for crt in seq crts do
  | CR.crt.index
end for
\end{verbatim}

The actions of this rule control the events related to the critical regions. The \texttt{RUN()} function can always communicate any desired member of the set of events passed as argument. The definition of \texttt{RUN} is depicted in Rule \cite{A.4}. Here we use an auxiliary function \texttt{ev()} that should return the set of events of the InteractionFragment passed as argument. Thus, the first \texttt{RUN()} carries out (in any order and in any number of times), the events in the set defined by the difference between the alphabet
of the events of the whole diagram and the events of the critical region. It may
be interrupted by the event beginCR.crt.index, which leads to a second run() that
only performs the events of the critical region. When the event endCR.crt.index
occurs, the action restarts liberating the other events to happen. Finally, the action
CRITICAL groups all other actions related to critical regions. It runs in parallel
with the rest of diagram (as can be seen in Rule [6.5]) to control the execution of
flows in and out of critical regions. This means that as soon as a critical region
starts (beginCR), this auxiliary action will restrict the events that may be per-
formed by the diagram, allowing only those events of the critical region. Once
this region ends (endCR), the auxiliary action comes back to the mode where all
other events outside of the region may happen.

The translation of the lifelines involved in the critical region adds the event beginCR.
right before the content of the critical region and the event endCR.$i right after.
What restricts the execution is the controller action (CRITICAL) described before.
Such action runs in parallel with the lifelines synchronising on all message events
of the enclosing fragment of the critical region.

**Rule 6.21**
\[
\begin{align*}
\text{t_critical_combined_fragment}(\text{crt: CombinedFragment, lf: Lifeline}) : \text{action} &= \\
\text{beginCR.crt.index \rightarrow Skip;}
\text{t_lf_interaction_fragments(crt.Operand.InteractionFragmentsFromLifeline(lf), lf)}
\text{;endCR.crt.index \rightarrow Skip}
\end{align*}
\]

In the next section we present the translation for state invariant constructor.

### 6.2.5 State Invariant

Rule 6.22 shows the function for translating state invariants. According to the
semantics, it is a runtime constraint that affects the validity of the traces; a trace
is valid if the invariant is evaluated to true, otherwise the trace is invalid. Therefore,
the rule is simple, as the action that it generates only collects state from
attributes and checks the constraint. When the evaluation returns false, then the
event invalid_trace is added to the trace meaning that it is not a valid one. During
analysis we can verify if such event is not inside the set of traces generated to
assure validity of the traces.
Rule 6.22 \( t_{\text{state invariant}}(si: \text{InteractionFragment}, lf: \text{Lifeline}): \text{action} = \)

\[
t_{\text{statecopy}}(lf.\text{represents}, si) \\
\text{if } si.\text{InteractionConstraint}.\text{specification} \text{ then Skip} \text{ else invalid_trace } \rightarrow \text{Skip} 
\]

6.2.6 InteractionUse

The Rules 6.23, 6.24, 6.25, and 6.26 for InteractionUse are divided in two groups: one for translation of the reference itself and its relationship with the enclosing diagram, and another for the translation of the lifelines that have an InteractionUse among their collections of InteractionFragments.

When we encounter an InteractionUse along the translation of a lifeline we should translate it according to the Rule 6.23. This rule only generates two events, one for firing the starting of the reference (\(\text{startRef}\)) and another for waiting until such reference finishes (\(\text{endRef}\)). How such events relate to the model of the diagrams is discussed next in the presentation of a set of rules that defines the translation of sequence diagrams that have InteractionUse elements.

Rule 6.23 \( t_{\text{interaction use}}(iu: \text{InteractionFragment}): \text{action} = \)

\[
\text{startRef}.iu.\text{index} \rightarrow \text{endRef}.iu.\text{index} \rightarrow \text{Skip}
\]

A set of rules is needed when defining a sequence diagram with InteractionUse elements. The root function is defined by Rule 6.24

Rule 6.24 \( t_{\text{complex sd}}(sd: \text{Interaction}): \text{seq of program paragraph} = \)

\[
\text{for } iu \text{ in seq sd.\text{InteractionUse} do} \\
\text{if } iu.\text{hasInteractionUse()} \text{ then} \\
\quad t_{\text{complex sd}}(iu) \\
\text{else} \\
\quad t_{\text{simple sd}}(iu, false) \\
\text{end if} \\
\text{t_create_iu_processes}(iu) \\
\text{end for} \\
\text{t_simple sd}(sd, true) \\
\text{t_create_complex_sd}(sd)
\]

155
First of all it translates all Interactions related to InteractionUse elements inside itself. The next step is to create the CML process that represents the InteractionUse elements (t_create_iu_processes), which is discussed in Rule 6.25. Then, we invoke one of the first rules discussed in this chapter (Rule 6.1), which creates the CML process for the enclosing diagram. However, this time we create it as an internal process (second argument of the function). This happens because such a process is reused by the next function (t_create_complex_sd() from Rule 6.26), which combines all other processes previously created in a parallel.

Rule 6.25 starts by defining two auxiliary processes that are responsible for indicating the start (startRef) and the end (endRef) of the InteractionUse execution. They are needed because we cannot compose actions with other processes in a CML process definition. Next the InteractionUse is defined. It receives as arguments the identifier of the InteractionUse, followed by the references (IDs) from the lifelines of the enclosing diagrams and references for gates. Although we use the term "gate", actually, it means a reference to lifeline that communicates some message through the gate. Finally, any arguments that it may need are defined as parameters of the process. The process itself is simply the sequential composition of the STARTREF process, followed by the process of the Interaction referred by the InteractionUse and, at last, the ENDREF process.
Rule 6.25 $t\_create\_i\_u\_processes(iu:\ InteractioUse):$ seq of program paragraph =

```
process \_STARTREF\_iu.index = begin @
    startRef.iu.index -> Skip
end
process \_ENDREF\_iu.index = begin @
    endRef.iu.index -> Skip
end

process InteractionUse\_iu.index =
    iu_id: ID;
for lf in seq iu.refersTo.Lifelines do
    t\_lifeline\_name(lf).id: ID;
end for
for g in seq iu.actualGates do
    name(g).id: ID;
end for
for arg in seq iu.Arguments
    name(arg): t\_types(arg.type);
endfor
    @ \_STARTREF\_iu.index; sd\_name(iu.refersTo)(iu_id, endfor
    @ for lf in seq iu.refersTo.Lifelines do
        t\_lifeline\_name(lf).id
end for
    @ for g in seq iu.actualGates do
        name(g).id
end for
    @ for arg in seq iu.Arguments
        name(arg)
endfor); \_ENDREF\_iu.index
```

Rule 6.24 shows how all of the CML processes created are arranged. It defines the root process, which is a parallelism of its internal version, which is defined by Rule 6.1, with all InteractionUse processes created earlier, which are arranged in an interleaving parallelism. For each InteractionUse process we use some functions to match the lifelines, gates and arguments. These functions are not defined here, but they simply determine those lifelines that are used inside the InteractionUse, the others that are not inside it, but communicate through gates, and, which arguments of the root Interaction must be passed as arguments of the Inter-
actionUse, respectively.

The alphabet of synchronisation are the startRef and endRef events added by the message events that communicate through gates. We assume that such events are returned by the function gateMessages(sd), because such events depends of the existence of gates in the InteractionUse.

Regarding the communication through gates, in order to make the message from the actual gate be the same that will be handled by the InteractionUse in its formal gate we need to perform some renaming. The function renameGateMessages(iu) does this work and it is detailed in Rule 6.27. Another assumption is that the message from outside the InteractionUse knows the lifeline linked inside the InteractionUse.
Rule 6.26  \( t\_create\_complex\_sd(sd: Interaction): seq of program paragraph = \)

\[
\text{process } \text{sd_name}(sd) = \text{sd_id}: ID;
\text{for } \text{lf} \text{ in seq } sd.\text{Lifelines do}
\text{t_lifeline_name(lf)}\_id: ID;
\text{end for}
\text{for gate in seq } sd.\text{formalGates do}
\text{name(gate)}\_id: ID;
\text{end for}
\text{for } m \text{ in seq } sd.\text{Messages do}
\text{for } p \text{ in seq } m.\text{Arguments do}
\text{if } p.\text{isInteractionParameter()} \text{ and notCreatedYet(p.name) then}
\text{p.name}: \text{t_types(p.type)} ;
\text{aux.add(p.name)}
\text{end if}
\text{end for}
\text{end for}
\text{for } iu \text{ in seq } sd.\text{InteractionUse do}
\text{InteractionUse\_iu.index (iu.index,}
\text{matchLifelines(sd,iu), matchGates(sd, iu), matchArguments(sd, iu)}
\text{renameGateMessages(iu)}\text{end for}
\text{end for}
\]

Rule 6.27 shows how to create the renaming mechanism to allow the synchroni-
sation of messages from formal gates to actual gates. After finding the actual gate corresponding to the formal gate, it checks the direction of the message. According to the direction and the type of the message, this function replaces the message id of the formal by the id of the actual message.

**Rule 6.27** renameGateMessages(iu: InteractionUse):

```
seq of program paragraph =
```

```
for formGate in seq iu.refersTo.formalGates do
for actualGate in seq iu.actualGates do
if name(formGate) == name(actualGate) then
  if formGate.Message.sendEvent == formGate then
    switch (formGate.Message.messageSort)
      case synchCall or reply:
        name(formGate.Message.receiveEvent.Lifeline.represents).op.
        formGate.Message.id <-
        name(formGate.Message.receiveEvent.Lifeline.represents).op.
        actualGate.Message.id
      case asynchCall:
        formGate.Message.id <-
        actualGate.Message.id
  end switch
else
  switch (formGate.Message.messageSort)
    case synchCall or reply:
      name(actualGate.Message.receiveEvent.Lifeline.represents).op.
      formGate.Message.id <-
      name(actualGate.Message.receiveEvent.Lifeline.represents).op.
      actualGate.Message.id
    case asynchCall:
      formGate.Message.id <-
      actualGate.Message.id
  end switch
end if
end if
end for
end for
```
To illustrate an example of application of these rules, we consider the two diagrams described in Section 6.1 regarding InteractionUse (figures 6.2 and 6.3).

The correspondent CML translation is shown in the next extracts. First we show the translation of the Interaction referred by the Interaction Use. This translation follows a similar idea of a regular Interaction as described in the previous sections. The only difference is that as it has a message arriving from a gate, we need to receive the identification of the sender as a parameter of this Interaction. Note that now there is a parameter in the process for the gate (gate3_id). We assume that it has the number 3 as identifier.

channels
  join, break, interrupt: nat;
  strict: nat.nat;
  beginCR, endCR: nat;
  startRef, endRef: nat;
  endInteraction: ID;
  invalid_trace, block: ID;

process sd_partial = sd_id:ID; Consumer_id: ID; Buffer_id: ID;
  gate3_id: ID; @ begin

chansets
  Hidden = {join, break, invalid_trace, endInteraction,
             strict, interrupt, beginCR, endCR!}
  cs_Consumer = ...
  cs_Buffer = ...
  Events = cs_Consumer union cs_Buffer

actions

  lf_Consumer = ((Buffer.op.2.Consumer_id.Buffer_id!
                mk_Buffer_types'rem_I() ->
                Buffer.op.2.Consumer_id.Buffer_id?
                out: Buffer_types'rem_O) -> Skip))

  lf_Buffer = ((Buffer.sig.5.gate3_id.Buffer_id?
              signal: Buffer_types'add_S -> Skip);
              (Buffer.op.2.Consumer_id.Buffer_id?
                oper: Buffer_types'rem_I -> Skip);
              (Buffer.op.2.Consumer_id.Buffer_id?
                out: Buffer_types'add_O) -> Skip))

@ ( (lf_Consumer [ cs_Consumer || cs_Buffer] lf_Buffer ))
  \ Hidden
end
Next we show the creation of the InteractionUse processes and the internal enclosing diagram. We assume that the identifier of the InteractionUse is the number 4.

First we have the definition of processes responsible for representing the starting and the ending of the InteractionUse. Next, the process for the InteractionUse itself, which sequentialise the process of the reference start ($\text{STARTREF}_4$), followed by the Interaction of the InteractionUse($\text{sd}_\text{partial}$), then the process of the reference end ($\text{ENDREF}_4$). The last process is the internal representation of the enclosing diagram, which synchronises the channels for the start ($\text{startRef}._4$) and end ($\text{endRef}._4$) of the reference execution.

$$
\begin{align*}
\text{process} \ \text{STARTREF}_4 &= \begin{align*} \text{begin } & @ \ \text{startRef}._4 \rightarrow \text{Skip} \ \text{end} \end{align*} \\
\text{process} \ \text{ENDREF}_4 &= \begin{align*} \text{begin } & @ \ \text{endRef}._4 \rightarrow \text{Skip} \ \text{end} \end{align*} \\
\text{process} \ \text{InteractionUse}_4 &= \begin{align*} \text{iu}_\text{id}: \text{ID}; \ \text{Consumer}_\text{id}: \text{ID}; \\
& \ \text{Buffer}_\text{id}: \text{ID}; \ \text{gate}_3\text{id} \ \text{ID}; \ \text{@} \\
& \ \text{STARTREF}_4; \ \text{sd}_\text{partial}(\text{iu}_\text{id}, \text{Consumer}_\text{id}, \text{Buffer}_\text{id}, \text{gate}_3\text{id}); \ \text{ENDREF}_4 \\
\text{process} \ \text{sd}_\text{internal}_\text{add}_\text{rem}_\text{ref} &= \begin{align*} \text{sd}_\text{id}: \text{ID}; \ \text{Producer}_\text{id}: \text{ID}; \\
& \ \text{Consumer}_\text{id}: \text{ID}; \ \text{Buffer}_\text{id}: \text{ID}; \ \text{item}: \ \text{Item}; \ \text{@} \ \text{begin} \\
& \text{chansets} \\
& \begin{align*} \text{Hidden} &= \{|\text{join}, \text{break}, \text{invalid_trace}, \text{endInteraction}, \text{strict}, \text{interrupt}, \text{beginCR}, \text{endCR}| \end{align*} \\
& \ \text{cs}_\text{Producer} = ... \\
& \ \text{cs}_\text{Consumer} = ... \\
& \ \text{cs}_\text{Buffer} = ... \\
& \ \text{Events} = \text{cs}_\text{Producer} \ \text{union} \ \text{cs}_\text{Consumer} \ \text{union} \ \text{cs}_\text{Buffer} \\
\text{actions} \\
& \ \text{lf}_\text{Producer} = ((\text{Buffer}.\text{sig}.1.\text{Producer}_\text{id}.\text{Buffer}_\text{id}! \text{mk}_\text{Buffer}_\text{types}.'\text{add}_S(\text{item}) \rightarrow \text{Skip})) \\
& \ \text{lf}_\text{Consumer} = ((\text{startRef}._4 \rightarrow \text{endRef}._4 \rightarrow \text{Skip})) \\
& \ \text{lf}_\text{Buffer} = ((\text{startRef}._4 \rightarrow \text{endRef}._4 \rightarrow \text{Skip})) \\
\end{align*} \\
\text{@} \\
\begin{align*} & (\text{lf}_\text{Consumer} \ [ \ \text{cs}_\text{Consumer} || \ \text{cs}_\text{Producer} \ \text{union} \ \text{cs}_\text{Buffer}] \\
& \ (\text{lf}_\text{Producer} \ [ \ \text{cs}_\text{Producer} || \ \text{cs}_\text{Buffer} ] \ \text{lf}_\text{Buffer} )) \\
& \ \backslash \ \text{Hidden} \\
\end{align*} \\
\text{end} \\
\end{align*}
$$

Finally, we present the complete enclosing diagram that organizes the defined process in parallel. Basically, it defines a generalized parallelism between its internal representation ($\text{sd}_\text{internal}_\text{add}_\text{rem}_\text{ref}$) and the InteractionUse ($\text{InteractionUse}_4$) processes. Here the gate parameter of the InteractionUse is
the ID of the Producer lifeline. Moreover, a renaming occurs because of the gate of the add asynchronous message. Then, instead of having 5 by id, the message of the formal gate will have 1 by id (Buffer.sig.5 <- Buffer.sig.1).

```plaintext
process sd_add_rem_ref = sd_id: ID; Producer_id: ID;
Consumer_id: ID; Buffer_id: ID; item: Item; @
sd_internal_add_rem_ref(sd_id, Producer_id, Consumer_id,
Buffer_id, item)
[[{|startRef, endRef,
    Buffer.sig.1.Producer_id.Buffer_id|}|
(InteractionUse_4(4, Consumer_id, Buffer_id, Producer_id)
[[Buffer.sig.5 <- Buffer.sig.1]])
```

### 6.3 Related work

In this section, we describe existing works in formal semantics of Sequence Diagrams.

There are numerous works providing semantics for Interaction Diagrams; most of them are related to UML instead of SysML. As they have similar semantics, most of the work done for UML fits well for SysML Sequence Diagrams. Usually there are two kinds of approaches: (i) the definition of a semantic model to formalise diagrams [LLJ04, SVN08], and (ii) the translation to an existing formalism such as Z, B, CSP, and Petri-Nets [EFM05, RW05, DD10]. The main advantage of the latter is the existing tool support used to apply reasoning on the translations. Few related works allow the check of the consistency among diagrams. This is extremely relevant to the COMPASS project as one of our objectives is to provide such consistency verificiation across the various SysML diagrams.

A Formal Universal Systems Semantics for SysML is proposed by Hamilton et al. [HHM+07] based on axioms of a general systems theory of the Universal Systems Language, 001AXES. Some mappings are provide for a subset of SysML diagrams, however Sequence Diagrams are not covered.

Storrle [Sto04] presents an exhaustive work on formalising Sequence Diagrams using trace semantics. Many constructs used in UML 2, including combined fragments, are covered. Storrle’s semantics allows one to reason about refinement, concurrency and time restrictions. Some basic features are not covered such as Gates and arguments. Haugen et al. [HHR+05] present another work that covers some of the Sequence Diagram elements we are interested in. They also propose an approach based on a trace semantics in which refinement is used as a founda-
tion for compositional analysis, verification and testing. Lund [Lun07] gives and operational semantics for the Haugen’s denotational semantics. In both semantics, loop with constraint and the BREAK fragment are not covered.

Although the approaches by Storrle [Sto04] and by Haugen et al. [HHRS05] do not use a semantic model similar to CML, they are inspiring works as some important discussions over complete and partial traces, global versus local view, and so on.

Dan and Danning [DD10] present an approach to semantic mapping specified using QVT [OMG05] relations to CSP [Hoa85]. Their approach uses a notation similar to CML, so we could benefit from some of the ideas of their work. However, very few constructs of UML 2 are covered.

Cavarra and Filipe [CKF05] proposed a technique using Object Constraint Language (OCL) templates to express liveness properties in UML Sequence Diagrams. They give examples showing that certain liveness properties cannot be expressed with assert or negate. Abstract state machines are used to enrich the sequence diagram in order to express such properties.

Cengarle [CMmuM05] gives an operational semantics for Sequence Diagrams. The author defines a semantic of negative fragments. Rules are given for each of the operators specifying whether a trace positively or negatively satisfies a fragment with that operator. The authors point out that with the basic interpretation of negative fragments it is easy to construct overspecified Interactions, i.e., an Interaction that can be positively and negatively satisfied from the same trace.

Knapp and Wuttke [KW07] provide an operational semantics based on automaton, while Eicher et al. [EFM05] use multivalued nets, which are a specific kind of Petri nets that allow parametrisation of messages and interactions. However, the latter does not provide enough information about the formalisation of some constructs. The intuitions are only described textually.

Shen et al. [SVN08] propose a formalization using template semantics is proposed for UML 2 Sequence Diagrams. The approach gives an operational semantics for which the basic computation model is hierarchical transition systems (HTS).

Most of these works differ regarding three main aspects: the number of constructions they cover, the semantics of constructions whose official meaning is vaguely defined, and the semantic domain used to formalise the semantics. The interesting aspect to notice is that when defining a semantics for Sequence Diagrams, some semantic decisions must be taken in order to allow its use. Micksei and Waese-lynck [MW11] have provided an excellent survey on these semantic choices. We
Table 6.1: Coverage comparison

<table>
<thead>
<tr>
<th></th>
<th>IU</th>
<th>GD</th>
<th>PA</th>
<th>ST</th>
<th>AL</th>
<th>OP</th>
<th>LP</th>
<th>BK</th>
<th>CR</th>
<th>SI</th>
<th>As</th>
<th>Gt</th>
</tr>
</thead>
<tbody>
<tr>
<td>[Sto04]</td>
<td>✓</td>
<td>×</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>×</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>[HHRS05]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>×</td>
<td>×</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>[CKF05]</td>
<td>×</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>[DD10]</td>
<td>×</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>×</td>
<td>×</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>[CMmuM05]</td>
<td>×</td>
<td>×</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>[KW07]</td>
<td>×</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>×</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>[EFM'05]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>[SVN08]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>×</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>×</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

developed our work according to their classification and categorisation of semantic meanings for sequence diagrams. Whenever the meaning of an operator is vaguely defined, we have chosen one that is more convenient for modelling and for checking diagram consistency.

As far as we researched none of these works aim to perform consistency verification among structural and behavioural diagrams.
Chapter 7

Activity diagrams

In this chapter, we formalise the translation of activity diagrams and illustrate it with the support of a few examples. Whenever convenient, we use the example of the Producer-Consumer introduced in the previous chapters. As this example does not illustrate all translation rules we propose, some other examples are also introduced along this chapter. This chapter is organised as follows. In Section 7.1, we introduced an initial example. The translation rules are presented in Section 7.2. Finally, we discuss related work in Section 7.3.

7.1 Examples

We depict two examples of activity diagrams, where they describe operations of the Buffer block. The operations are \textit{add} and \textit{rem}. These diagrams can be invoked by the state machine or other activity diagram in order to fulfill the behaviour of such tasks.

The first diagram displayed in Figure 7.1 represents the \textit{Add} activity. It has all kinds of activity nodes: action, control and object nodes. The beginning of its behaviour starts through the initial node and the activity parameter node \textit{item}, which is of the type \textit{Item}. A decision node checks if the size of the buffer is less than five (limit of the Buffer), so another item can be added, otherwise the activity ends. When it is less than five, an opaque action is fired and it receives as input through an input pin the item to be added. The content of the opaque action is CML code, which is executed when it can begin its processing. Such code just adds to the buffer the element received as parameter. After this action is performed, the control edge leaving this action leads to an activity final node,
which ends the diagram execution.

![Activity Diagram: Adding an element to the Buffer](image)

**Figure 7.1: Activity Diagram: Adding an element to the Buffer**

The *Rem* activity diagram removes an item from the buffer and returns it as output. Figure 7.2 shows the behaviour of such activity. It has no inputs, and after the initial node, a decision node checks if there is at least one element to be removed, otherwise it ends the activity. In case it has elements, a call operation action is fired, which should deal with the task of removing an element of the buffer and returning it as output. In this case, such task may be carried out by the block itself, where the task may be described using an Action language, or the state machine may address this operation call. This call operation action returns an output, which is put in its output pin. Next, this data is made available as output of the activity by means of the activity parameter node *Rem* of the type *Item*. The two edges leaving the action represent an object flow that transports the data to the activity parameter node, and the other is a control flow that leads to the end of the diagram at an activity final node.

We use these examples to illustrate how the translation functions generate the respective CML code. SysML activities have their semantics based on token-flow semantics of Petri nets. Tokens are used to hold values of input, output, and control that flow from one action to another [FMS11]. To deal with control flow, we will use a channel that is responsible for synchronisation between actions.
7.2 Formal semantics

In this section we present the translation rules for activity diagrams. We use a CML process to model an activity diagram. A process for all activity diagrams of a block combines in parallel the processes for each activity diagram. Channels are used for synchronisation of internal actions. These channels are hidden from the outside the CML action for an activity diagram. For instance, a channel that is used to model control flow is hidden from the outside. We use channels to model activity parameter nodes. CML actions model control nodes of a SysML activity diagram. Interruptible regions are also modelled as CML actions. Object nodes are described by means of actions that involve external choice or interleaving of actions with communication through channels. The main action of the process that models an activity diagram is defined by a recursive action composed by a sequential composition of three components. The first component is the action that models how an activity starts. It is followed by a parallel action involving the actions that models all nodes of a diagram and the action that model how an activity finishes. The third component is the communication of values after the activity finishes.

We first present the translation rule for activity diagrams of a block (Section 7.2.1), then we focus on the translation of a single activity (Section 7.2.2). We define rules for Call Behaviour Action parallelism, and internal activity diagram (a definition we need that is related to Call Behaviour Action) in Section 7.2.3 and 7.2.4.
respectively. We introduce the rules for translating how an activity starts and how an activity ends in Sections 7.2.5 and 7.2.7, respectively. The for rule interruptible regions is presented in Section 7.2.6. The rules for actions, control nodes, and object nodes are defined in Sections 7.2.8, 7.2.9, and 7.2.10, respectively.

7.2.1 Block Activity Diagrams

We start the translation using the function \( t_{activity\_diagrams} \) defined in Rule 7.1. This is the root function for activity diagrams. The parameters of this function provide a sequence of activity diagrams and a block to which these activities belong (when they define operations of the block), resulting in a sequence of program paragraphs. This function introduces channels related to control flow, and a process.

The channel \( \text{control} \) is used to deal with the flow of the control tokens in an activity diagram. The execution of an action or control node linked to a control flow is only allowed with the availability of the control token. The type of the channel \( \text{control} \) is \( \text{nat} \). Similarly to sequence diagrams, we assume a pre-processing of an activity diagram so that each action node is given a unique identifier of type \( \text{nat} \). So, through \( \text{control} \) we establish which action node can be executed. Channel \( \text{endDiagram} \) is used to communicate that a diagram with an ID has finished its execution because it reached a final activity node or, if a final activity node is not present in a diagram, after writing output parameters or after sending signals. The type of channel \( \text{endDiagram} \) is defined by the activity identifier. The channel \( \text{endFlow} \) is used for indicating that we have executed an action that ends a flow, for instance, a Send Signal action without outgoing edges. The other cases that end a flow are defined by a flow final node, and output activity parameters, which only have incoming edges. Channel \( \text{interrupted} \) is used to indicate that an interrupting edge of an interruptible region has been traversed. The interruption is related to a diagram, a block instance, and an interruptible region index. All these channels are internal to an activity diagram process and they will be hidden by using a specific function to be presented later in this chapter.

By calling the function \( t_{ad\_channels(ad, block)} \) for every activity diagram in the sequence \( ads \) received as argument, more channels are introduced to deal with communication that occurs among diagrams and inside diagrams. Also, the iterative call to function \( t_{activity\_diagram} \) provides a translation of every activity diagram in \( ads \).

The name of the process that the function \( t_{activity\_diagrams} \) introduces is defined by the name of the block instance received as argument (\( \text{name(block)} \)) appended
with _ads. The process definition is given by the alphabetised parallelism given by a call to function define_alphabetised_parallel with the following arguments: the activity diagrams of block and a set of synchronisation events determined by the alphabet (of visible channels) of each diagram given by function alpha(ad).

**Rule 7.1**

\[
\text{t_activity_diagrams(ads: seq of Activity, block: Block): seq of program paragraph =}
\]

\[
\begin{align*}
\text{channels} \\
\text{control: nat;}
\text{endDiagram: ID;}
\text{endFlow: ID.nat;}
\text{interrupted: ID.ID.nat;}
\end{align*}
\]

\[
\text{for ad in ads do}
\text{\quad t_ad_channels(ad, block)}
\text{end for}
\]

\[
\text{for ad in ads do}
\text{\quad t_activity_diagram(ad, block)}
\text{end for}
\]

\[
\begin{align*}
\text{process name(block)_ads = name(block)_id: ID @}
\define_alphabetised_parallel\{((\text{name(ad)}\ (\text{name(block)}_id), \alpha(ad)) | ad \in \text{seq ads})
\end{align*}
\]

The respective CML code generated by this rule for the Add and Rem activity diagrams described earlier is displayed in the following extract.

**channels**

\[
\begin{align*}
\text{control: nat;}
\text{endDiagram: ID;}
\text{endFlow: ID.nat;}
\text{interrupted: ID.ID.nat;}
\end{align*}
\]

\[
\begin{align*}
\text{process ad_Add = ...}
\text{process ad_Rem = ...}
\text{process Buffer_ads = Buffer_id: ID @}
\text{ad_Add(Buffer_id) [ {||} || {||} ] ad_Rem(Buffer_id)}
\end{align*}
\]

In this extract, there are two processes (ad_Add and ad_Rem) for the activities Add.
and Rem. The process Buffer_ads combines these two processes in parallel. The channels deal with control flow, indication of finalisation of a specific diagram, end of a flow, and indication that an interrupting edge has been traversed.

The function \( t_{ad} \text{channels} \) (Rule 7.2) introduces channels for a specific activity \( ad \) of a Block instance \( block \). To start an activity, we must provide values for its input parameter nodes by means of channel \( \text{startActivity} \). The types of parameters of this channel are defined to be ID followed by the types of the activity parameter nodes that do not have incoming edges, but have outgoing edges. In other words, they are used for input. After an activity finishes, it provides values by means of channel \( \text{endActivity} \) whose type is defined in a similar way to that of \( \text{startActivity} \). On the other hand, parameter nodes related to \( \text{endActivity} \) have no outgoing edge, but have at least one incoming edge. Notice that the channel names \( \text{startActivity} \) and \( \text{endActivity} \) are appended by an underscore (\(_\)) followed by the activity ID.

Similarly to an activity, a call behaviour action has channels for providing input values and for obtaining output values. The channel named \( \text{startActivity}_\text{CBA} \) is appended by the diagram unique name given by \( \text{id(ad)} \); its type is given by the diagram identifier, the Call Behaviour Action identifier (natural), and the types of the input activity parameter nodes. Channel \( \text{endActivity}_\text{CBA} \) is similar to \( \text{startActivity}_\text{CBA} \), but the former is to get values that result from a Call Behaviour Action execution.

Every object flow is related to a channel whose name is \( \text{obj} \) followed by a unique name given to the model by \( \text{id(ad)} \), and edge index through which the object flows. The type of this channel is given by the type of the object flow source.

We introduce channels for the input and output pins of each action node. Communication through an input pin uses a channel named \( \text{in} \) followed by the action name, the input pin edge index and \( \text{id(ad)} \). The type is given by the input pin type. Similarly, for communication through an output pin, we define channels named \( \text{out} \) followed by \( \text{id(ad)} \), the action name and the output edge index.
Rule 7.2 \( t\_ad\_channels(ad: \text{Activity}, block: \text{Block}): \text{seq of channel} = \)

\[
\begin{align*}
\text{startActivity}\_id(ad): & \; \text{ID} \{ \text{t}\_\text{types}(param) \mid \text{param in seq ad.Nodes(} \\
\text{ActivityParameterNodes.Type) and param.\text{IncomingEdges.size()} == 0 \text{ and} \\
\text{param.\text{OutgoingEdges.size()}} > 0\}; \\
\text{endActivity}\_id(ad): & \; \text{ID} \{ \text{t}\_\text{types}(param) \mid \text{param in seq ad.Nodes(} \\
\text{ActivityParameterNodes.Type) and param.\text{OutgoingEdges.size()} == 0 \text{ and} \\
\text{param.\text{IncomingEdges.size()}} > 0\};  \\
\text{for cba in seq ad.Nodes(CallBehaviour.Type) do}  \\
\text{startActivity}\_\text{CBA}\_id(ad): & \; \text{ID.\text{nat}} \{ \text{t}\_\text{types(edge)} \mid \text{edge in cba.\text{IncomingEdges(Object.Type)}} \};  \\
\text{endActivity}\_\text{CBA}\_id(ad): & \; \text{ID.\text{nat}} \{ \text{t}\_\text{types(edge)} \mid \text{edge in cba.\text{OutgoingEdges(Object.Type)}} \};  \\
\text{end for}  \\
\text{for objFlow in seq ad.\text{ActivityEdges do}} \\
\text{obj}\_id(ad)\_\text{edge.index}: & \; \text{t}\_\text{types(objFlow.source)};  \\
\text{end for}  \\
\text{for action in seq ad.Nodes(Action.Type) do}  \\
\text{for inPin in seq action.inputPins do}  \\
\text{in}\_id(ad)\_\text{name(action)}\_\text{inPin.edge.index}: & \; \text{t}\_\text{types(inPin)};  \\
\text{end for}  \\
\text{for outPin in seq action.outputPins do}  \\
\text{out}\_id(ad)\_\text{name(action)}\_\text{outPin.edge.index}: & \; \text{t}\_\text{types(outPin)};  \\
\text{end for}  \\
\text{end for}  \\
\end{align*}
\]

The respective CML code generated by Rule 7.2 regarding the additional channels of the Add and Rem activity diagrams is displayed in the following extract (we assume that the IDs of the diagrams are their own names):

```cml
channels
...
startActivity Add: ID.Item;
endActivity Add: ID;
obj Add 1: Item;
in Add OpaqueAction1 1: Item;

startActivity Rem: ID;
endActivity Rem: ID.Item;
```

172
In the next section, we introduce the function to translate a single activity diagram.

### 7.2.2 Single Activity Diagram

The function `t_activity_diagram` (Rule 7.3) gives as results two process definitions. The first process results from a call to `t_ad_internal_process` and is related to the nodes of an activity diagram of a block. The second process is named `ad_followed by the name of the activity diagram. It is parametrised by a block identifier. The process definition is given by a generalised parallelism of two processes: the one introduced by calling function `t_adInternal_process` and other one for Call Behaviour actions. When a Call Behaviour action is to be executed, the process that models the diagram that requests such execution must synchronise with the action model in the channel that allows the model of a Call Behavior action to start its execution. After this, the action that was called synchronises with the diagram process in a channel to indicate that it has finished. In this way, the diagram process gets back control.

**Rule 7.3**

\[
\text{t_activity_diagram}(\text{ad}: \text{Activity}, \text{block}: \text{Block}): \text{seq of program paragraph} =
\]

\[
\text{t_ad_internal_process}(\text{ad, block})
\]

\[
\text{process } \text{ad_name}(\text{ad}) = \text{name(block)}\_\text{id}: \text{ID}; \@
\]

\[
\text{ad_internal_name}(\text{ad})(\text{name(block)}\_\text{id})
\]

\[
\text{t_ad_cba_parallel}(\text{ad, block})
\]

Next, we illustrate in the following extract the application of Rule 7.3 by showing the generated CML code for the `Add` and `Rem` activity diagrams (as they do not have call behaviour actions they are reduced to their internal CML process):

```plaintext
channels

... 

process ad_internal_Add = ...

process ad_Add = Buffer_id: ID; @ ad_internal_Add(Buffer_id)

process ad_internal_Rem = ...
```
process ad_Rem = Buffer_id: ID; @ ad_internal_Rem(Buffer_id)

7.2.3 Call Behaviour Action Parallelism

We use the function \( t_{\text{ad_cba_parallel}} \) (Rule 7.4), if there is at least one Call Behaviour action in a diagram, to introduce channels used for synchronisation between the diagram model and call behaviour actions models. They run in parallel without synchronising. This function \( t_{\text{ad_cba_parallel}} \) has an activity identification and a block instance as arguments.

The channels whose names begin with `startActivity_CBA_` are used for synchronisation between a diagram model and call behaviour action (CBA) model, so that the CBA model starts its execution. The channel name is composed by the activity unique id given by \( \text{id}(\text{ad}) \), the block name appended with \_id, and the call behaviour action index. The channels whose names begin with `endActivity_CBA_` are used for synchronisation between a CBA action model and a diagram model, indicating the end of a CBA execution. Every CBA action model has these two channels.

**Rule 7.4**

\[
\text{if ad.Nodes(CallBehaviour.Type).size > 0}
\]

\[
\begin{align*}
&\{[\{\text{startActivity_CBA_id(ad).name(block)}_{\text{id}}.\text{cba.index}} \\
&\text{endActivity_CBA_id(ad).name(block)}_{\text{id}}.\text{cba.index} \\
&\text{cba in seq ad.Nodes(CallBehaviour.Type)}\}\}
\end{align*}
\]

\[
\begin{align*}
&\text{ad_name(cba_ad)} \\
&\{[\{\text{startActivity_id(cba_ad).name(block)}_{\text{id}}.\text{cba.index}} \\}
\end{align*}
\]

\[
\text{if ad.AdNodes(CallBehaviour.Type).size > 0}
\]

\[
\begin{align*}
&\{[\{\text{startActivity_id(cba_ad).name(block)}_{\text{id}}.\text{cba.index}} \\}
\end{align*}
\]

The CML actions for the Call Behaviour actions are executed in parallel without synchronising. The channel name that begins with `startActivity_` of each ac-
tion is renamed so that it gets the same name as used in the set of synchronising events. A similar renaming is done with respect to channels with names starting with `endActivity_`.

### 7.2.4 Internal Activity Diagram

To translate the nodes and edges of an activity, we use the function `t_ad_internal_process` from Rule 7.5 that expects an activity ID, and a block instance as arguments. The function introduces a process with name `ad_internal_` and appended with the activity name. This process expects two arguments. The first is an activity instance; the second is a block instance. If the block instance given as argument is not null, function `t_ad_internal_process` calls two functions to complete the translation: one for introducing channel sets, and another one for the translation of nodes of an activity diagram.

**Rule 7.5**

\[
t_{ad\_internal\_process}(ad: \text{Activity}, block: \text{Block}): \text{seq}\text{ of process} =
\]

\[
\text{process ad\_internal\_name(ad) =}
\]

\[
\text{if block} \neq \text{null then}
\]

\[
\quad \text{name(block)\_id: ID;}
\]

\[
\text{end if}
\]

\[
\quad @ \text{begin}
\]

\[
\quad t_{ad\_chansets}(ad, block)
\]

\[
\quad t_{ad\_actions}(ad, block)
\]

\[
\text{end}
\]

The next extract shows the application of Rule 7.5 regarding the `Add` activity diagram. The translation of `Rem` is a similar.

\[
\ldots
\]

\[
\text{process ad\_internal\_Add = Buffer\_id: ID; } @ \text{begin}
\]

\[
\quad \text{chansets}
\]

\[
\ldots
\]

\[
\quad \text{actions}
\]

\[
\ldots
\]

\[
\text{end}
\]

\[
\ldots
\]
The function \( t_{\text{ad-chansets}} \) (Rule 7.6) introduces the set of channels \( \text{Hidden} \) with all those channels that must not be visible outside a process, which are related to control flow (channels \( \text{control}, \text{endDiagram}, \text{endFlow}, \text{interrupted} \)), object flow (channels whose names start with \( \text{obj_} \)), input pins (channels whose name start with \( \text{in_} \)), and output pins (channels whose name start with \( \text{out_} \)). The chanset \( \text{Hidden} \) is used in a hiding operation in Rule 7.30.

**Rule 7.6**

\[
t_{\text{ad-chansets}}(\text{ad: Activity, block: Block}): \text{seq of action} =
\]

\[
\text{chansets} = \{|\text{control, endDiagram, endFlow, interrupted,}
\]

\[
\text{|}
\]

\[
\text{for ObjEdge in seq ad.edges(ObjectFlow.Type) do}
\]

\[
\text{obj_id(ad)\_ObjEdge.index}
\]

\[
\text{end for}
\]

\[
\text{for action in seq ad.Nodes(Action.Type) do}
\]

\[
\text{|for inPin in seq action.inputPins do}
\]

\[
\text{in_id(ad)\_name(action)\_inPin.edge.index}
\]

\[
\text{end for}
\]

\[
\text{|for outPin in seq action.outputPins do}
\]

\[
\text{out_id(ad)\_name(action)\_outPin.edge.index}
\]

\[
\text{end for}
\]

\[
|
\]

The next extract shows the application of Rule 7.6 to \( \text{Add} \).

... process ad\_internal\_Add = Buffer_id: ID; @ begin

\[
\text{chansets}
\]

\[
\text{Hidden} = \{|\text{control, endDiagram, endFlow, interrupted,}
\]

\[
\text{obj\_Add_1, in_add\_OpaqueAction_1|}
\]

\[
|\text{actions}
\]

\[
\text{|...}
\]

\[
|\text{end}
\]

... The function \( t_{\text{ad-actions}} \) (Rule 7.7) receives as arguments an activity and a block instance. The translation of activity actions is performed by the translation of actions of an activity diagram that belong to a block and by introducing a CML
main action. Notice that function \( t_{\text{ad\_actions}} \) just introduces CML actions, as indicated by the use of the reserved word actions.

**Rule 7.7** \( t_{\text{ad\_actions}}(\text{ad}: \text{Activity}, \text{block}: \text{Block}): \text{seq of action} = \)

\[
\text{actions}
\]

\[
\text{if block} \neq \text{null}
\]

\[
\quad t_{\text{block\_actions}}(\text{ad}, \text{block})
\]

\[
\text{end if}
\]

\[
\quad t_{\text{main\_ad\_action}}(\text{ad}, \text{block})
\]

To introduce CML actions that correspond to actions of an activity diagram, we call the function \( t_{\text{block\_actions}} \) (Rule 7.8) that receives an activity and a block instance as arguments. This function calls other functions to translate activity nodes.

The CML action \text{END\_DIAGRAM} communicates through channel endDiagram\_id(ad), indicating that this diagram has finished and then behaves as \text{skip}. Function \( t_{\text{start\_activity}} \) introduces a CML action that defines how an activity diagram starts its execution. We use the function \( t_{\text{interruptible\_regions}} \) to introduce models of interruptible regions. For each node in an activity diagram we select an adequate function based on the node type. There are separate functions for action nodes, control nodes, and object nodes. The action \text{Nodes} is defined as the alphabetised parallelism of all actions that correspond to the nodes of an activity diagram. According to the type of the node a different function is called to recover the alphabet of synchronisation of the node (chanset), \( t_{\text{chanset\_action}}(\text{node}) \) for actions, \( t_{\text{chanset\_control}}(\text{node}) \) for control nodes and \( t_{\text{chanset\_object}}(\text{node}) \) for object nodes. These functions should return all channels used by each node in their translations. For example, a control node, which has two edges, one incoming and the other an outgoing edge both from control flow with indexes 1 and 2, respectively, has by alphabet the following chanset: \{\text{control.1, control.2}\}. Finally, function \( t_{\text{end\_activity}} \) is called to introduce a CML action that establishes the finishing of an activity diagram.
Rule 7.8 \( t_{\text{block\_actions}}(ad: \text{Activity}, \text{block: Block}): \text{seq of action} = \)

\[
t_{\text{start\_activity}}(ad, \text{block})
\]
\[
\text{END\_DIAGRAM} = \text{endDiagram} \text{.id(ad)} \rightarrow \text{Skip}
\]
\[
t_{\text{interruptible\_regions}}(ad, \text{block})
\]

for node in seq ad.Nodes do

\[
\text{switch}(node\text{.Type})
\]

\[
case \text{Action.Type: } t_{\text{action\_node}}(node, ad, \text{block}, node\text{.inInterruptibleRegion})
\]
\[
case \text{ControlNode.Type: } t_{\text{control\_node}}(node, ad, \text{block}, node\text{.inInterruptibleRegion})
\]
\[
case \text{ObjectNode.Type: } t_{\text{object\_node}}(node, ad, \text{block}, node\text{.inInterruptibleRegion})
\]

end switch

end for

\[
\text{Nodes} = \text{define\_alphabetised\_parallel}\{
\]

\[
\text{for node in seq ad.Nodes}
\]

\[
\text{switch}(node\text{.Type})
\]

\[
case \text{Action.Type: } (name(node)\_node.index, t_{\text{chanset\_action}}(node, ad))
\]
\[
case \text{ControlNode.Type: } (CNode\_node.index, t_{\text{chanset\_control}}(node, ad))
\]
\[
case \text{ObjectNode.Type: } (ObjNode\_node.index, t_{\text{chanset\_object}}(node, ad))
\]

end switch

end for

\[
\}
\]
\[
\text{t\_end\_activity}(ad, \text{block})
\]

The next extract shows the application of Rules 7.7 and 7.8 to Add. We assume that the indices for the opaque action, initial node, decision node, the two activity final nodes, the input pin and the input activity parameter node are, respectively, 1, 1, 2, 3, 4, 1 and 2.

...
END_DIAGRAM = endDiagram.Add -> Skip

OPAQUEACTION_1 = ...
CNode_1 = ...
CNode_2 = ...
CNode_3 = ...
CNode_4 = ...
ObjNode_1 = ...
ObjNode_2 = ...

Nodes = (OPAQUEACTION_1 [ || || ]
        (CNode_1 [ || || ]
         (CNode_2 [ || || ]
          (CNode_3 [ || || ]
           (CNode_4 [ || || ]
            (ObjNode_1 [ || || ] ObjNode_2 )))))))

END_ACTIVITY = ...

...
The next extract shows the application of Rule 7.9 to Add. When we only have one CML action in the interleaving parallelism (|||), it can be reduced to such respective action, that is why we only have (item := x_item -> Skip) instead of the parallelism.

... process ad_internal_Add = Buffer_id: ID; @ begin chansets ...
... actions ...
    START_ACTIVITY = startActivity_Add.Buffer_id?x_item ->
                        atomic (item := x_item)
... end ...

### 7.2.6 Interruptible Regions

An interruptible region allows a subset of the action executions to be forbidden to flow through the diagram by the destruction of their tokens; the others remain executing. A mechanism called interrupting edge indicates when an interruptible region is to be interrupted: a token is accepted by the interrupting edge. We use function $t_{interruptible\_regions}$ (Rule 7.10) to translate interruptible regions. A CML action whose name begins with InterruptibleRegion_ and has as suffix the interruptible region index is introduced for every interruptible region of an activity diagram of a block instance. The CML action is recursive (defined using the µ construct). When an interrupting edge is to be traversed, the actions inside an interruptible region must finish execution. This is modelled by the external choice involving all interrupting edges that leave the region. The synchronisation should be accomplished via one of the events defined by control.edge.index or
those of channels with name beginning with \textit{obj\_}. Then, an event over a channel whose name begins with \textit{interrupted} is available to the environment, defining an interrupting edge of an interruptible region. This channel is specific to the model of a diagram, a block instance (\texttt{name(block)}\_id), and interruptible region (\texttt{intRegion.index}). After synchronisation, the CML action behaves as \texttt{skip}. The external choice is sequentially composed with the recursive call \texttt{x}, indicating that the region that was interrupted is now available to start execution again.

The CML action \texttt{InterruptibleRegions} is defined by the parallelism of all CML actions introduced for interruptible regions. As these regions execute asynchronously, they are in interleaving.

\textbf{Rule 7.10} \texttt{t\_interruptible\_regions(ad: Activity, block: Block): seq of action =}

\begin{verbatim}
for intRegion in seq ad.group and intRegion.isInterruptibleActivityRegion() do
    InterruptibleRegion\_intRegion.index = mu X @ (  
        [1] for edge in seq intRegion.interruptingEdge do
            if edge.isControl() then
                control.edge.index
            else
                obj\_id(ad)\_edge.index?x
            end if
        end for
        -> interrupted.id(ad).name(block)\_id.intRegion.index -> skip
    end for
)
end for
\end{verbatim}

\texttt{InterruptibleRegions =}

\begin{verbatim}
for intRegion in seq ad.group and intRegion.isInterruptibleActivityRegion() do
    InterruptibleRegion\_intRegion.index
end for
\end{verbatim}

\textbf{7.2.7 End Activity}

The function \texttt{t\_end\_activity} (Rule 7.11) introduces the CML action \texttt{END\_ACTIVITY}, which models the protocol to finish an activity execution. This situation may happen due to the execution of an activity final node, otherwise, when all flows end. This function is applied to an activity of a block instance (arguments). If there are activity final nodes in a diagram, the function introduces an external choice
of all activity final nodes actions. In the case of a control edge, synchronization is expected over channel `control`; otherwise it is an object flow, synchronization is expected over a corresponding channel name that begins with `obj_`. The interleaving finishes after synchronization over all channels (control and object flow). Then, the action behaves as `END_DIAGRAM`. If there are no activity final nodes, we establish that any node that can end a flow—Flow Final nodes, Send Signal nodes without output edges, and Activity Parameter nodes that only have input edges—must synchronize with an event over channel `endFlow` appended with the node index. These actions involving channel `endFlow` are executed in an interleaving that is sequentially composed with the action `END_DIAGRAM`. The function `ad.endFlowElements` returns all elements, as stated above, that can end a flow.

**Rule 7.11**

```
Rule 7.11 t_end_activity(ad: Activity, block: Block): seq of action =

END_ACTIVITY =
if ad.Nodes(ActivityFinalNode.Type).size > 0 then
    for fNode in seq ad.Nodes(ActivityFinalNode.Type) do
        for edge in seq fNode.IncomingEdges do
            if edge.Type == Control.Type then
                control.edge.index -> Skip
            else
                obj_id(ad)_edge.index?x_edge.index -> Skip
            endif
        end for
    end for
else
    for fNode in ad.endFlowElements() do
        endFlow.id(ad).fNode.index -> Skip
    end for
end if
```

The next extract shows the application of Rule 7.11 for `Add`.

```
process ad_internal_Add = Buffer_id: ID; @ begin chansets ...
```

182
The action `END_ACTIVITY` models that the activity `Add` finishes after synchronisation on events `control.3` or `control.4`.

### 7.2.8 Actions

The function `t_action_node` Rule 7.12 deals with the translation of an action node. It receives as arguments an action node to be translated, the activity diagram to which the action belongs, a block instance from which we translate the diagram, and the diagram interruptible regions. This function introduces a CML action with a name composed by the SysML action name appended with an underscore and the action node index.

In SysML, every action execution finishes when the executing diagram finishes execution. If the action is part of an interruptible region, however its execution finishes when the interrupting edge of the region to which it belongs accepts a token. Notice that it is possible to have nested interruptible regions, so that when an interruptible region finishes its execution, actions of any nested interruptible region must also finish execution. The boolean variable `end_guard` indicates when an action can have its execution finished. This variable is used as the guard of a guarded action: if it is true, then the action after the guard is enabled. So, for every action, function `t_action_node` declares `end_guard`, a boolean variable, with `true` as initial value.

The CML action for a SysML action is recursive. If there are any input or output pins from an action, we declare variables to hold the values communicated by input pins or values that are going to be used in output pins. The variable identifiers are defined by the names of the input and output pins or by object nodes. The type of these variables is defined by the type of the input and output pins or types of object.

If there are incoming edges to an action (control and data), we translate them by using the function `t_action_incoming_edges` applied to the action and the diagram. Then, the function `t_action_types(action,ad,block,regions)`, is called to translate the
action according to its type. All the previous CML actions are sequentially composed with other CML actions that are related to the outgoing edges. If they exist, then we make a recursive call to enable the action to deal with other incoming request. Otherwise, the node is a end of flow, then it can end the flow or receive another request (recursive call). Notice that the variables initially declared are visible through the whole action.

A SysML action can have its execution finished due to two situations: (1) an interrupting edge has accepted a token (and the action is part of the interrupting region) or (2) diagram execution has finished. If the action belongs to at least one interruptible region, then we insert an interruption in a guarded action. The guard is the boolean variable \texttt{end\_guard} that, if true, enables an action defined as the external choice over the indices of interruptible regions of a diagram, using channel \texttt{interrupted} appended with the diagram identifier, block name and \texttt{\_id.i}, where \texttt{i} is the interruptible region index. If an interrupting edge accepts a token, there will be an event that synchronises with one of the channels available for external choice, then we have a recursive call.

Independently of the existence of interruptible regions, SysML actions finish when an executing activity to which the actions belong finishes. This is defined by the action \texttt{end\_guard \& END\_DIAGRAM} that takes control only when the guard \texttt{end\_guard} value is \texttt{true}, then it behaves as \texttt{END\_DIAGRAM}.
Rule 7.12 \( t\text{\textunderscore action\_node}(\text{action: ActivityNode, ad: Activity, block: Block, regions: seq of InterruptibleActivityRegion}): \text{action} = \)

\[
\text{name(\text{action})\_action.index} = (\text{dcl end\_guard: bool} = \text{true} @ \text{mu X @} \\
\text{if action\_input.size()} > 0 \text{ or action\_output > 0} \\
\text{dcl [] \{name(obj): t\_types(obj); \} obj in seq concat(action\_input, action\_output) or} \\
\text{obj in \{inObj | edge in action.IncomingEdges(Object.Type) and} \\
\text{inObj == edge.source} \text{ or} \\
\text{obj in \{outObj | edge in action.OutgoingEdges(Object.Type) and} \\
\text{outObj == edge.target\}} \\
\text{@ \}
\]

\text{end if}

\text{if action\_incoming\_edges.size > 0 then}
\text{t\_action\_incoming\_edges(action, ad)}
\text{end if}

\text{(t\_action\_types(action, ad, block, regions))};

\text{if action\_outgoing\_edges.size > 0 then}
\text{t\_action\_outgoing\_edges(action, ad) X)}
\text{else}
\text{if action\_outgoing\_edges.size == 0 then}
\text{endFlow.id(ad).action.index -> Skip [] X )}
\text{end if}
\text{end if}

\text{if regions.size() > 0 then}
\text{/\\ end\_guard & \{ \{intRegion\_index | intRegion in seq regions\}\} @}
\text{interrupted.id(ad).name(block\_id.i -> X}
\text{end if}
\text{/\\ end\_guard & END\_DIAGRAM)}

The function \( t\text{\textunderscore action\_types} \) (Rule 7.13) receives as arguments an action node, the activity diagram to which the node belongs, a block instance, and the diagram interruptible regions. Based on the type of the activity node, this function selects an adequate function for translation. The following types of actions can be translated: call operation, opaque, accept event, send signal, value specification, and call behaviour.
Rule 7.13 \( t\_action\_types\) (action: ActivityNode, ad: Activity, block: Block, regions: seq of InterruptibleActivityRegion): action =

```
switch(action.Type)
  case CallOperation.Type: t\_call\_operation\_action(action,ad,block)
  case Opaque.Type: t\_opaque\_action(action,ad,block)
  case AcceptEvent.Type: t\_accept\_event\_action(action,ad,block,regions)
  case SendSignal.Type: t\_send\_signal\_action(action,ad,block)
  case ValueSpecification.Type: t\_value\_specification\_action(action,ad,block)
  case CallBehaviour.Type: t\_call\_behaviour\_action(action,ad,block)
end switch
```

An action only starts its execution if all data it needs and the control token are available. This is specified as an interleaving of control edges and data input. Concerning control, there is an interleaving involving the control channel, each one synchronising on the incoming control edge index. After synchronisation, the action behaves as \texttt{Skip}. The interleaving of the input data is accomplished on channels with name prefixed with \texttt{in} appended with the following elements separated with an underscore: the activity unique identifier given by \texttt{id(ad)}, the action name, and the incoming edge index. The input parameter names (which are preceded by a question mark) of these channels begin with \texttt{in} appended with the edge index. After receiving a value over a channel whose name starts with \texttt{in}, such value is assigned to the variable that has already been introduced by function \texttt{t\_action\_node}. After both interleavings of the actions just described finishes, the CML action for the activity node can continue as there is a sequential composition with the following the action described by the function \texttt{t\_action\_node}.

Rule 7.14 \( t\_action\_incoming\_edges\) (action: ActivityNode, ad: Activity): action =

```
{ control.inEdge.index -> \texttt{Skip} \mid inEdge in seq action.IncomingEdges(Control.Type)}
\{ in\_id(ad)\_name(action)\_inObj.edge.index inObj.name(inObj) -> name(inObj):= inname(inObj) \mid inObj in seq action.input or inObj in \{obj \mid edge in action.incomingEdges(Object.Type) and obj == edge.source\} \}
```

Function \texttt{t\_action\_outgoing\_edges} (Rule 7.15) defines how a SysML action that has
not been interrupted deals with passing the control to other activity nodes and with data output. The result from this function is sequentially composed with other actions as described in function $t_{action\ node}$. The function $t_{action\ outgoing\ edges}$ receives as arguments an action node and an activity. This function introduces an interleaving of control edges and another for dealing with output. First, it attempts to synchronize over channel $control$ with the outgoing edge indices as parameter. After synchronization, the CML action behaves as $Skip$. Output of values is accomplished by using channels whose names start with $out$ appended with the unique name for an activity ($id(ad)$), the action name ($name(action)$), and the outgoing edge index ($outObj.edge.index$). This channel communicates the value of variable $name(outObj)$, the outgoing edge name.

**Rule 7.15** $t_{action\ outgoing\ edges}(action: \ ActivityNode, ad: \ Activity): \ action =$

\[
( \begin{align*}
&control.outEdge.index \rightarrow \text{Skip} \quad \text{outEdge in seq} \\
&\text{action.OutgoingEdges(Control.Type)} \\
&\text{out}_id(ad)\_name(action)\_outObj.edge.index \text{!name(outObj)} \rightarrow \text{Skip} \\
&\text{outObj in seq action.output or} \\
&\text{outObj in \{obj | edge in action.OutgoingEdges(Object.Type) and} \\
&\text{obj == edge.target \}}) \}
\]

**CallOperationAction**

A CallOperationAction is an action that an operation call to a target (block). The function $t_{call\ operation\ action}$ (Rule 7.16) receives as arguments an activity node ($action$) of an activity diagram ($ad$) of a block instance ($block$). This action attempts to synchronize with a block operation over a channel with name that begins with the name of the block, given by $name(action.operation.class)$, followed by an operation identifier. The channel expects an operation call identifier ($m_id$); it outputs the sender block name $name(block)$ appended with and underscore, and the receiver block name. The last parameter is related to the operation data. We use $mk_$ to create the operation data. Notice that the data source is given by input pin names. After calling an operation of another block, we assign $end\ guard$ the value $false$, indicating that the executing action cannot be interrupted. We wait the return from the operation call on a channel with a name that begins with the block operation owner name ($name(action.operation.class)$). The channel definition is followed by an operation identifier, the operation call identifier name appended with $_id$, and sender block name also appended with $_id$, and the receiver block name. The input parameter $oper$ receives values returned from the operation.
The values are assigned to output pins; after each assignment the action behaves as `skip`. Only after calling and returning from an operation, we establish that the executing action can be interrupted, and assign `true` to `end_guard`. Finally, the action behaves as `skip`.

**Rule 7.16**

```plaintext
rule _call_operation_action(action: ActivityNode, ad: Activity, block: Block):
  action =

  name(action.operation.class).op?m_id!name(block)_id!name(action.target)
  !mk_name(action.operation.class)_types?name(action.operation)_I(
    [name(inPin) | inPin in seq action.input and inPin != action.target]
  ) -> end_guard := false;

  name(action.operation.class).op?m_id!name(block)_id!name(action.target)
  ?oper: name(action.operation.class)_types'
    name(action.operation)_O -> (  
      [name(outPin) := oper.ret -> skip  
        | outPin in seq action.output and ret in seq action.operation.OutputParameters]  
    ); end_guard := true
```

The next extract shows the application of Rules 7.12–7.16 for the CallOperation-Action of the `Rem` activity diagram.

```plaintext
... process ad_internal_Rem = Buffer_id: ID; @ begin
... chansets
... actions
...  rem_1 = (dcl end_guard: bool = true @ mu X @ (  
    dcl rem:Item; @
    (control.3 -> Skip);  
    (Buffer.op?m_id!Buffer_id!Buffer_id!mk_Buffer_types’rem_I() ->  
      end_guard := false;  
      Buffer.op?mi_id!Buffer_id!Buffer_id?oper: Buffer_types’rem_O ->  
      rem := oper._ret ; end_guard := true;  
    (control.4 -> Skip ||| out_Rem_rem_1!rem -> Skip); X)  
    \ end_guard & END_DIAGRAM)
... end
...
```

In this extract, we establish that after data is available for calling an operation and the operation is called, it is not possible to interrupt the action `rem_1`. This is established by the assignment of `false` to `end_guard`. After the operation returns
a value, the action `rem_1` can be interrupted. After this, the action attempts to synchronise with a control event and output the value received on the channel that begins with `out_`.

**OpaqueAction**

The function `t_opaque_action` (Rule 7.17) defines the translation for an activity node (`action`) of an activity diagram (`ad`) of an instance block (`block`). The CML action for a SysML opaque action is defined by the SysML action body (`action.body`).

```plaintext
Rule 7.17 t_opaque_action(action: ActivityNode, ad: Activity, block: Block): action =
    action.body
```

The next extract shows the application of Rules 7.12-7.17 for the OpaqueAction of the `Add` activity diagram.

```plaintext
...
process ad_internal_Add = Buffer_id: ID; @ begin
chansets ...
actions
    ...
    OPAQUEACTION_1 = {dcl end.guard: bool = true @ mu X @ {
        dcl x: Item; @
        (control.2 -> Skip || in_Add_OpaqueAction_1?inx -> x := inx);
        (Buffer.get_b.OpaqueAction.Buffer_id?b -> b := b^[x] ->
         Buffer.set_b.OpaqueAction.Buffer_id!b -> Skip);
        (control.4 -> Skip); X)
    /
    end.guard & END_DIAGRAM)
    ...
end ...
```

**SendSignalAction**

Using the function `t_send_signal_action` (Rule 7.18) we can translate an activity node (`action`) that is a SysML Send Signal action of an activity diagram (`ad`) from a block instance. The CML action is defined by a communication and then
the action behaves as \texttt{skip}. The communication occurs over a channel with a name that begins with the signal target type name, followed the indication that we are dealing with a signal (.\texttt{sig}). We expect a signal call identifier as input. An output is the sender block name appended with \_id; another output is the receiver block name. Any data sent is based on input pins.

\textbf{Rule 7.18}  

\begin{verbatim}
Rule 7.18 t_send_signal_action(action: ActivityNode, ad: Activity, block: Block):
    action =
        name(action.target.Type).sig?m_id! name(block)_id! name(action.target)
          !mk_name(action.target.Type)_types' name(action.signal)_S ( 
            \{ name(inPin) \mid inPin in seq action.input and inPin != action.target \}
          ) -> Skip
\end{verbatim}

\section*{ValueSpecificationAction}

A ValueSpecificationAction is an action that evaluates a value specification [Obj11]. Function \texttt{t_value_specification_action} translates an activity node (\texttt{action}) of an activity diagram (\texttt{ad}) that is related to a block instance (\texttt{block}). The activity node, a value specification action, is defined as an assignment to a variable with name given by \texttt{name(action.result)}, an output pin.

\textbf{Rule 7.19}  

\begin{verbatim}
Rule 7.19 t_value_specification_action(action: ActivityNode, ad: Activity, block: Block):action =
    name(action.result):= action.value
\end{verbatim}

\section*{AcceptEventAction}

The function \texttt{t_accept_event_action} (Rule 7.20) receives the following as arguments: an activity node (\texttt{action}), which is a accept event action, a activity diagram (\texttt{ad}), a block instance (\texttt{block}), and sequence of interruptible regions (\texttt{regions}). This function returns a CML action. An Accept Event action with no incoming edges located in an interruptible region only starts execution when the interruptible region starts execution. If there are interruptible regions and an accept event action received as argument has no incoming edge, we introduce an external choice involving the incoming edges to the interruptible regions. These edges
have source outside the region. These edges can be related to a control flow or object flow. These are defined by channels control or a channel with name beginning with obj_. After synchronisation, the action behaves as skip. Only after termination of the external choice, the accept event action is enabled to receive a signal. This is established by the sequential composition. A signal is received over a channel with name beginning with the receiver block name, followed by a signal identifier. The channel expects a message identifier (m_id) and a signal sender identifier (sender). It outputs the receiver block name name(block) appended with _id. Any object is received on parameter signal, then we assign it to output pins of the accept event action. Then, the action behaves as skip.

**Rule 7.20**

```plaintext
t_accept_event_action(action: ActivityNode, ad: Activity, block: Block, regions: seq of InterruptibleActivityRegion): action =
if regions.size() > 0 and action.IncomingEdges.size == 0 then
    for region in seq regions do
        for edge in seq region.edges do
            if notContains(edge.source, region) then
                if edge.isControl() then
                    control.edge.index -> Skip
                else
                    obj_id(ad)_edge.index?x_edge.index -> Skip
                end if
            end if
        end for
    end for
end if

name(block).sig?m_id?sender!name(block)_id
?signal:name(block)_types’ name(action.trigger.event)_S -> {
    |[11] [name(outPin) := signal.ret
    | outPin in seq action.output and ret in seq action.trigger.event.Parameters]}
```

**CallBehaviourAction**

The function stated in Rule 7.21 receives as arguments an activity node (cba), which is a call behaviour action, an activity (ad), and a block instance (block).
The CML action synchronizes over a channel whose name begins with `startActivity_CBA_`, appended with the diagram identifier. This channel synchronizes with an event of a specific block name (given by `name(block)`)) and the node index under translation. The output parameters of this channel are defined by the names of the incoming edges sources. Then, we assign `false` to the variable `end_guard`, stating that action cannot be interrupted. Afterwards, the action expects an event to synchronize over the channel with a name that begins with `endActivity_CBA_`, appended with the diagram identifier. This channel synchronizes with an event of a specific block name (given by `name(block)`)) and call behaviour action index (in the diagram). The input parameters are defined by the names of targets of the outgoing edges of the call behaviour action node. Finally, we assign `true` to variable `end_guard`, stating that the action can be interrupted.

**Rule 7.21**

\[
t_{\text{call\_behaviour\_action}}(cba: \text{ActivityNode}, ad: \text{Activity}, block: \text{Block}): \text{action} = \]

\[
\text{startActivity}_\text{CBA}_\text{id(ad),name(block)}_\text{id}.cba.index
\]

\[
\{ \text{name(obj)} \mid \text{obj in \{\text{edge.source | edge in cba.IncomingEdges(Object.Type)\}} \} \}
\]

; end\_guard := \text{false}

\[
\text{-} \text{endActivity}_\text{CBA}_\text{id(ad),name(block)}_\text{id}.cba.index
\]

\[
\{ \text{name(obj)} \mid \text{obj in \{\text{edge.target | edge in cba.OutgoingEdges(Object.Type)\}} \} \}
\]

; end\_guard := \text{true}

In the next section, we introduce the functions to translate control nodes

### 7.2.9 Control Nodes

The function \( t_{\text{control\_node}} \) (Rule 7.22) defines the translation of control nodes. The translation of each control node introduces a CML action with a name that begins with `CNode_` followed by the node index (\( \text{ctr.index} \)). Based on the type of the control node, we call the appropriate function. A control node can have one of the following types: initial, flow final, decision, merge, fork, and join. If there are interruptible regions, a control action can be interrupted if it is inside an interruptible region and the interrupting edge accepts a token; the action must finish. The action that models the control node can be interrupted by an action that offers an external choice. The environment can choose one of the events over a channel with name `interrupted` appended with the diagram identifier, the block identifier, and an interruptible region index i. If an interrupting edge accepts a token, the environment (interrupting edges) synchronizes over one of
these channels, finishing the control action execution. Then, the action behaves as the control node action again (named CNode_ appended with the control node index). When a diagram finishes, all executing actions must finish, they are all interrupted by the action END_DIAGRAM.

**Rule 7.22**

\[ \text{t\_control\_node}(\text{ctr: ActivityNode, ad: Activity, block: Block, regions: seq of InterruptibleActivityRegion}): \text{action} = \]

\[
\text{CNode\_ctr.index} = \{ \}
\text{switch(ctr.Type)}
\text{case Initial.Type: t\_initial\_node(ctr,ad,block)}
\text{case FlowFinal.Type: t\_initial\_node(ctr,ad,block)}
\text{case Decision.Type: t\_decision\_node(ctr,ad,block)}
\text{case Merge.Type: t\_merge\_node(ctr,ad,block)}
\text{case Fork.Type: t\_fork\_node(ctr,ad,block)}
\text{case Join.Type: t\_join\_node(ctr,ad,block)}
\text{end switch}
\text{if regions.size() > 0 then}
\text{\[ \{\text{intRegion.index | intRegion in seq regions}\} \] @}
\text{interrupted.id(ad).name(block).id.i \rightarrow CNode\_ctr.index}
\text{end if}
\text{\} END\_DIAGRAM}

**InitialNode**

The function \( t\_initial\_node \) (Rule 7.23) defines how to translate initial nodes (\( \text{ctr} \)) of an activity diagram (\( \text{ad} \)) of a block instance (\( \text{block} \)). The function introduces interleaved actions that communicate control!x, where \( x \) is the index of an outgoing edge of an initial node.

**Rule 7.23**

\[ t\_initial\_node(\text{ctr: ActivityNode, ad: Activity, block: Block}): \text{action} = \]

\[
\| | | | x: \{ \text{x.index | x in seq r.OutgoingEdges} \}
\text{\} @ control!x \rightarrow \text{Skip}
\]

The next extract shows the application of Rules 7.22 and 7.23 to the InitialNode of the \textit{Add} activity diagram. Since it only has one outgoing edge, the set that
defines the interleaving in Rule 7.23 only has one element, which is the index of the control edge (the number 1 in this case).

... process ad_internal_Add = Buffer_id: ID; @
... chansets ...
... actions ...
... CNode_1 = (||| x:{1} @ control!x -> Skip) \ END_DIAGRAM ...
...

**FlowFinalNode**

The function \( \text{t_flow_final_node} \) (Rule 7.24) receives as argument a control node \( \text{ctr} \), an activity \( \text{ad} \), and a block instance defined by the parameter \( \text{block} \). This function introduces an interleaving of actions related to every incoming edge to a flow final node. In the case of a control flow, the channel control is used for synchronisation, and it behaves as \( \text{Skip} \). For object flows, a channel with name starting with obj_, which is appended with the diagram unique identifier given by \( \text{id(ad)} \) and the edge index, expects a value on parameter _edge.index. Then it behaves as \( \text{Skip} \). After the interleaving of actions finishes, there is an external choice between events endFlow.id(ad).ctr.index, where \( \text{ctr.index} \) is the index of the control node under translation, and the action that models the control node introduced by the Rule 7.22 (CNode_appended with the control node index).

**Rule 7.24** \( \text{t_flow_final_node(ctr: ActivityNode, ad: Activity, block: Block): action =} \)

\[
\{ \ | | |
\text{for edge in seq ctr.IncomingEdges do}
\text{if edge.Type == Control.Type then}
\text{control.edge.index -> Skip}
\text{else}
\text{obj_id(ad)_edge.index?x_edge.index -> Skip}
\text{end for}
\}; (endFlow.id(ad).ctr.index -> Skip []) CNode_ctr.index
\]
**DecisionNode**

The function $t_{\text{decision\_node}}$ (Rule 7.25) establishes how to translate a decision node. If the incoming edge is concerned with control, then the action expects synchronisation over channel $\text{control}$ with the control incoming edge index as parameter. Notice that this rule uses the auxiliary rule $t_{\text{statecopy}}(\text{block, ctr})$ that creates a local copy of the block instance state. A non deterministic if statement is introduced with the guard of each guarded command defined by the guards of the outgoing edges from the decision node. This is followed by an event over channel $\text{control}$ with the decision node outgoing edge index. In the case of an object flow, a non deterministic if statement is introduced with one guarded command for each guard of the outgoing edges. When a guard is satisfied, the object received via the channel that represents the incoming edge of the decision node is communicated through a channel named $\text{obj}_\text{appended}$ with the following: the diagram unique identifier (provided by $\text{id(ad)}$), an underscore, the outgoing edge index $\text{outEdge.index}$. 
Rule 7.25 \( t_{\text{decision node}}(\text{ctr}: \text{ActivityNode}, \text{ad}: \text{Activity}, \text{block}: \text{Block}): \text{action} = \)

\[
\begin{align*}
\text{if} & \ \text{ctr.IncomingEdge.Type == Control.Type then} \\
& \quad \text{control.ctr.IncomingEdge.index } \rightarrow \text{t.statecopy(block, ctr)} \\
& \quad \text{if} \ \text{outEdge in ctr.OutgoingEdges do} \\
& \quad \quad \text{if} \ \text{outEdge.guard == "else" then} \\
& \quad \quad \quad \text{not (\{x.guard | x in ctr.OutgoingEdges and x != outEdge\})} \\
& \quad \quad \quad \quad \rightarrow \ \text{control.outEdge.index } \rightarrow \ \text{CNode ctr.index} \\
& \quad \quad \text{else} \\
& \quad \quad \quad \text{outEdge.guard } \rightarrow \ \text{control.outEdge.index } \rightarrow \ \text{CNode ctr.index} \\
& \quad \text{end if} \\
& \quad \text{end for} \\
& \text{else} \\
& \quad \text{obj_id(ad) ctr.IncomingEdge.index?x ctr.IncomingEdge.index) } \rightarrow \text{t.statecopy(block, ctr)} \\
& \quad \text{if} \ \text{outEdge in ctr.OutgoingEdges do} \\
& \quad \quad \text{if} \ \text{outEdge.guard == "else" then} \\
& \quad \quad \quad \text{not (\{x.guard | x in ctr.OutgoingEdges and x != outEdge\})} \\
& \quad \quad \quad \quad \rightarrow \ \text{obj_id(ad) outEdge.index!name(ctr.IncomingEdge.selection.output)} \\
& \quad \quad \quad \quad \rightarrow \ \text{CNode ctr.index} \\
& \quad \quad \text{else} \\
& \quad \quad \quad \text{outEdge.guard } \rightarrow \ \text{obj_id(ad) outEdge.index!name(ctr.IncomingEdge.selection.output)} \\
& \quad \quad \quad \quad \rightarrow \ \text{CNode ctr.index} \\
& \quad \text{end if} \\
& \quad \text{end for} \\
& \text{end if}
\end{align*}
\]

The next extract shows the application of Rules 7.22 and 7.25 to the DecisionNode of the Add activity diagram.

\[
\ldots
\text{process } \text{ad_internal_Add = Buffer_id: ID; @ begin} \\
\ldots
\text{chansets} \\
\ldots
\text{actions} \\
\text{\quad CNode_2 = (control.1 } \rightarrow \text{ dcl b:Item @} \\
\]

196
The action CNode_2 models the decision node that appears in Add. Depending on the buffer size, this action synchronizes on the event control.3 or control.2. Then, the action behaves as CNode_2 again. This action can be interrupted if the activity Add finishes.

MergeNode

A merge node is translated by the function t_merge_node (Rule 7.26). It receives as arguments an activity node (ctr), which is a merge node, an activity (ad), and a block instance (block). The incoming edges of a merge node are all either of object flows or all of control flows. If the outgoing edge of a merge node is a control flow (Control.Type), then the action that models the merge node synchronizes with one of the incoming control edges from the set of indexes given by \{x.index | x in ctr.IncomingEdges\}. Then the action communicates over the control channel the outgoing edge index. Finally, the action behaves again as the (merge) control node. In the case of object flow, the action offers to the environment the possibility of communication over channels with name starting with obj_ for each incoming edge. The whole channel name is defined by the concatenation of obj_ with id(ad)_, and the incoming edge index (inEdge.index). The input parameter is called x. After communication of a value recorded in x, such value is given as output over a channel whose name starts with obj_ appended with the with id(ad)_, and the outgoing edge index (ctr.OutgoingEdge.index). After output, the action behaves again as the merge control node action just introduced.
Rule 7.26 \( t\_merge\_node(ctr: \text{ActivityNode}, \ ad: \text{Activity}, \ block: \text{Block}): \text{action} = \)

\[
\text{if } ctr.\text{OutgoingEdge}.\text{Type} == \text{Control}.\text{Type} \text{ then}
\]
\[
\quad \text{control!i:}\{\{x.\text{index} \mid x \text{ in } ctr.\text{IncomingEdges}\}\} \to
\]
\[
\quad \text{control.ctr.\text{OutgoingEdge}.index} \to \text{CNode}_\text{ctr.index}
\]
\[
\text{else}
\]
\[
\quad [] \text{ for } \text{inEdge } \text{in seq ctr.\text{IncomingEdges}} \text{ do}
\]
\[
\quad \quad \text{obj}_\text{id(\text{ad})_\text{inEdge}.\text{index}}?x \to \text{obj}_\text{id(\text{ad})_\text{ctr.\text{OutgoingEdge}.index}}!x
\]
\[
\quad \quad \to \text{CNode}_\text{ctr.index}
\]
\[
\quad \text{end for}
\]
\[
\text{end if}
\]

ForkNode

A fork node has just one incoming edge that is an object flow or a control flow. The function \( t\_fork\_node (\text{Rule}\ 7.27) \) introduces an action for a fork node. In the case of an incoming control flow, the channel \text{control} is used for synchronization with the channel that represents the incoming edge index. Then, the control token is available for all target actions of the outgoing edges from the fork node. Subsequently to the communication with each control channel of outgoing edges, the CML action behaves like \text{Skip} . In the case of object flow, the (fork) action synchronizes with the object incoming edge through channel \text{obj}_\_, appended with the incoming edge index, receiving an object in the parameter \( x \). After this, the actions behave like an interleaving to communicate the object \( x \) through each channel \text{obj}_\_ (appended with the outgoing edge index \text{outEdge/index}) . After the object is consumed by each outgoing edge, the (fork) action starts again. The sequential composition guarantees that only after finishing the execution of the interleaving related to control or object flow, the whole action behaves again as itself .
Rule 7.27 \( t_{\text{fork-node}}(\text{ctr}: \text{ActivityNode}, \text{ad}: \text{Activity}, \text{block}: \text{Block}): \text{action} = \)

\[
\text{if } \text{ctr.IncomingEdge.Type} == \text{Control.Type} \text{ then } \\
\quad \text{control.ctr.IncomingEdge.index} \rightarrow \\
\qquad (|||i:\{x.index \mid x \in \text{ctr.OutgoingEdges}\} \otimes \text{control!i} \rightarrow \text{Skip};) \\
\quad \text{CNode_ctr.index} \\
\text{else } \\
\quad \text{obj_id(ad)_ctr.IncomingEdge.index?x} \rightarrow \\
\qquad (||| \text{for outEdge in seq ctr.OutgoingEdges do } \\
\qquad \text{obj_id(ad)_outEdge.index!x} \rightarrow \text{Skip} \\
\qquad \text{end for } ) \text{; CNode_ctr.index} \\
\text{end if}
\]

JoinNode

The function \( t_{\text{join-node}} (\text{Rule 7.28}) \) receives as arguments an activity node \( \text{ctr} \), which is the join node, an activity diagram \( \text{ad} \), and a block instance \( \text{block} \). For each object incoming edge of a join node, in our model we declare a variable with name beginning with \( \text{edge}_\text{index} \) and appended with the edge index. The type of each variable is the same as the type of the edge source in SysML. For each incoming edge related to control flow, we introduce an action prefixed with the \text{control} channel synchronising on the edge index. Then the action behaves as \text{Skip}. If the incoming edge is related to object flow, we introduce an action that begins with an event over a channel whose name begins with \text{obj}_\text{index} and expects a value on a parameter with name \( x_\text{index} \) followed by the edge index. Then the object is assigned to one of the variables declared in the beginning of the action. We distinguish variables according to the edge index. After all communication via channels that model (control and object) incoming edges, we check if there is any object flow into the join node; notice that communication is sequentially composed with the remaining of the action. If there is an object flow, we use channel \text{obj}_\text{index} (appended with the diagram unique identifier, an underscore, and outgoing edge index) to communicate an object. If there are only control incoming nodes to the join node, we attempt to synchronise with the control outgoing edge index over channel \text{control}. After communication in both cases, the action behaves as itself.
Rule 7.28 \textit{t\_join\_node}(ctr: ActivityNode, ad: Activity, block: Block): action =

\begin{verbatim}
(dcl
  for edge in seq ctr.IncomingEdge(Object.Type) do
    edge_edge.index: t\_types(edge.source)
  end for

  let objIndex = -1 in
  for inEdge in seq ctr.IncomingEdges do
    if inEdge.Type == Control.Type then
      control.inEdge.index -> Skip
    else
      objIndex = inEdge.index
      obj\_id(ad)_inEdge.index\_x\_inEdge.index ->
      edge\_inEdge.index := x\_inEdge.index -> Skip
    end if
  end for

  if objIndex != -1 then
    obj\_id(ad)_ctr.OutgoingEdge.index\_edge\_objIndex) -> CNode\_ctr.index
  else
    control!ctr.OutgoingEdge.index) -> CNode\_ctr.index
  end if
end let
\end{verbatim}

7.2.10 Object Nodes

The function \textit{t\_object\_node} (Rule 7.29), translates an object node into CML. It receives an object node, an activity, and a block instance block as arguments. The function introduces an action with a name that begins with \texttt{ObjNode\_} followed by the object node index. The action is recursive.

We deal with three kinds of object nodes: input or output pin, input or output activity parameter node and simple object node. The definition of an action for an input pin is defined by a communication over a channel with a name that begins with \texttt{obj\_}; the remaining of the channel name is composed by the diagram unique identifier, and the object incoming edge. The input parameter of this channel is named \texttt{x\_} appended with the object index. Then, via a channel that models an in-
put pin we communicate $x_\_appended with the object index. After the occurrence of synchronisation events, we have a recursive call.

In the case of an output pin, the sequence of events begins with a communication via the channel that models the output pin, followed by a communication through a channel that models an object node. After the occurrence of both synchronisation events, we have a recursive call.

In the case of activity parameter nodes, which serve as output for an activity (they have no outgoing edges but have at least one incoming edge), we define an external choice over channels that model the incoming edge to the parameter node. The channels names begin with $obj\_ appended with a diagram unique identifier $(id(ad))$, an underscore, the incoming edge index $(obj.IncomingEdge.index)$, an underscore, the incoming object index $(obj.index)$, and the output parameter $x_\_appended with the object index. Then, the action assigns the value received after communication to a variable with the same name as the object. After assignment, the action behaves as an external choice between an event $(endFlow)$ and a recursive call.

When the activity parameter is an input, we define an interleaving of actions; each one defined by a communication followed by $skip$. If the object node is not a pin, or an activity parameter node, the model is defined by an input parameter $x_\_appended with the object index, of the channel with name beginning with $obj_\_appended with a diagram unique identifier $(id(ad))$, an underscore, and the incoming edge index $(obj.IncomingEdge.index)$. Then, the action attempts to output the value received through a channel with a name similar to the one used for input; the value in $x_\_appended with the object index is used, is used for output.

If an object node is inside an interruptible region its execution finishes if a token is accepted in an interrupting edge. The action that models this case can choose one of the events over a channel with name $interrupted appened with the diagram id, the block id $(name(block)\_id)$, and an interruptible region index $i$. If an interrupting edge accepts a token, the environment defined by interruptible regions actions will synchronise over one of these channels, finishing the object node action execution. Then, the action will behave as the object node action again. When a model of an activity finishes, all executing object node actions must finish, they are all interrupted by the action $END\_DIAGRAM$. 

201
Rule 7.29 \( t \_object\_node(obj: \text{ActivityNode}, ad: \text{Activity}, block: \text{Block}): \text{action} = \)

\[
\text{ObjNode}_\_\text{obj.index} = \mu X @ (\{
\text{switch(obj.Type)}
\text{case InputPin.Type:}
\quad \text{obj}_\text{id(ad)} \_\text{obj.IncomingEdge.index}?x_\text{obj.index} \rightarrow
\quad \text{in}_\text{id(ad)} \_\text{name(obj.OwnerNode)} \_\text{obj.IncomingEdge.index}!x_\text{obj.index} \rightarrow X)
\text{case OutputPin.Type:}
\quad \text{obj}_\text{id(ad)} \_\text{obj.OutgoingEdge.index}!x_\text{obj.index} \rightarrow X)
\text{case ActivityParameterNode.Type:}
\quad \text{if obj.OutgoingEdges.size == 0 and obj.IncomingEdges > 0 then}
\quad \text{for edge in seq obj.IncomingEdges do}
\quad \quad \text{obj}_\text{id(ad)} \_\text{edge.index}?x_\text{obj.index}
\quad \quad \rightarrow \text{name(obj)} := x_\text{obj.index} \rightarrow
\quad \quad (\text{endFlow.id(ad)} \_\text{obj.index} \rightarrow \text{Skip} [ ] X)
\quad \text{else}
\quad \quad \text{for edge in seq obj.OutgoingEdges do}
\quad \quad \text{obj}_\text{id(ad)} \_\text{edge.index}!\text{name(obj)}
\quad \quad \rightarrow \text{Skip);X})
\text{end if}
\text{default:}
\quad \text{obj}_\text{id(ad)} \_\text{obj.IncomingEdge.index}?x_\text{obj.index} \rightarrow
\quad \text{obj}_\text{id(ad)} \_\text{obj.OutgoingEdge.index}!x_\text{obj.index} \rightarrow X)
\text{end switch}
\text{if regions.size() > 0 then}
\quad \text{\#/i: \{intRegion.index | \text{intRegion in seq regions}\} \emptyset}
\quad \text{interrupted.id(ad)} \_\text{name(block)} \_\text{id.i} \rightarrow X
\text{end if}
\}) \#/ \text{END_DIAGRAM}

The next extract shows the application of Rule 7.29 to the input pin of the Opaque-Action and the input activity parameter node of the \text{Add} activity diagram.

\[
\ldots
\text{process ad_internal_Add = Buffer_id: ID; @ begin}
\text{chansets}
\ldots
\text{actions}
\quad \text{ObjNode}_1 = \mu X @ (}
\]

202
The function \( t_{\text{main\_action}} \) (Rule 7.30) introduces the main CML action for an activity diagram. It receives an activity and a block instance as arguments. The main action is recursive, it is composed of a block declaration (for variables) and an action. The variables have the same name as the activity parameter nodes that are used to hold activity input or output values. The types of these variables are defined as the types of the parameter nodes. These variables are assigned default values defined by the function \( \text{default}(t_{\text{types}}(\text{param})) \).

The action begins with \( \text{START\_ACTIVITY} \), which represents the beginning of the diagram and has as start event the channel \( \text{startActivity} \). This action is sequentially composed by a generalised parallelism of actions \( \text{Nodes} \) and \( \text{END\_ACTIVITY} \), and interruptible regions (if they exist in the diagram). The \( \text{Nodes} \) action synchronises with the \( \text{END\_ACTIVITY} \) action through the channels used by the CML action \( \text{END\_ACTIVITY} \), which should come from function \( t_{\text{chanset\_end\_activity}}(\text{ad}) \). This function may simply traverse the activity final nodes and nodes that end the flow to gather the set of events of the type \( \text{control}.* \) and \( \text{obj}_.* \) that leads to an activity final node plus \( \text{endFlow}.* \) events. The same happens for the \( \text{InterruptibleRegions} \) action, however, it has another function to get the chanset of synchronisation \( t_{\text{chanset\_int\_regions}}(\text{ad}) \), which follows the same idea, that is, return the channels used in the interruptible region actions.

This parallelism runs until a diagram finishes its execution. Then, the main action communicates over the channel named \( \text{endActivity} \), appended with the diagram unique identifier, and the block name appended with \_id. The output data of this channel is defined by the names of the activity parameter nodes that have no outgoing edges, just incoming edge. After the whole action execution termination, the \( \text{START\_ACTIVITY} \) is made available again. Notice that we use the hiding operation \( \backslash \) to hide internal channels in set \( \text{Hidden} \), so that they are not visible outside the main action.
Rule 7.30  \( t_{\text{main\_action}}(ad: \text{Activity}, \text{block}: \text{Block}): \text{action} = \)

\[
@ \mu X @ (\text{dcl} \{ \text{name(param)}: \text{t\_types(param)} := \text{default(t\_types(param))} \} @ \text{START\_ACTIVITY}; (\text{Nodes} \{ |\{\text{t\_chanset\_end\_activity}\text{(ad)}\} | \} \text{END\_ACTIVITY}) \text{if} \text{ad.InterruptibleRegions.size} > 0 \text{then} \{ |\{\text{t\_chanset\_int\_regions}\text{(ad)}\} | \} \text{InterruptibleRegions}; \text{else} \{ \text{endActivity\_id}(ad).\text{name(block)}\_id \} \{ |\{\text{name(param)} \mid \text{param in seq ad.Nodes(ActivityParameterNodes.Type)} \} | \text{param.IncomingEdges.size()} == 0 \text{and} \text{param.IncomingEdges.size()} > 0 \} \}; X \ \text{Hidden}
\]

Finally, the next extract details the application of Rule 7.30, which represents the main action of activity diagram. Here we depict the two main actions of each one of the activity diagrams Add and Rem. Notice that the Add diagram has an input activity parameter node, and then a local variable is created (item, same as the name of the node) to store the input data. The Rem diagram returns an item, which is represented by the local variable of the same name of the node, as can be seen in the output activity parameter node from Figure 7.2.

\[
\text{process ad\_internal\_Add = Buffer\_id: ID; @ begin} \\
\text{chansets} \\
\text{actions} \\
\text{...} \\
\text{...} \\
\text{mu X @ (dcl item: Item := null @} \\
\text{START\_ACTIVITY}; (\text{Nodes} \{ |\{\text{control.3, control.4}\} | \} \text{END\_ACTIVITY}); \text{endActivity\_Add.Buffer\_id} \}; X \ \text{Hidden} \\
\text{end} \\
\text{...} \\
\text{process ad\_internal\_Rem = Buffer\_id: ID; @ begin} \\
\text{chansets} \\
\text{actions} \\
\text{...}
\]
This finishes the description of translation rules for activity diagrams.

### 7.3 Related work

In this section we describe some previous work about the formalisation of activity diagrams.

Xu et al. \[X^{+08}, X{\text{M}}^{09}\] formalise UML activity diagrams and define a set of mapping rules from the formal model for activity diagrams into CSP \[\text{Hoa85}\]. They introduce a formal meta-model for activity diagrams. This meta-model is given by a tuple composed of elements that represent the different nodes of an activity diagram, a set directed edges, and the flow relationship between them. Translation functions are defined for each of the their translation is not compositional, the diagram nodes are not translated independently. Some of their mapping differs from ours. For instance, in their work a decision node is mapped into a guarded event. In our translation, we use Dijkstra’s guarded commands for the translation of a decision node. The reason is that if two or more guards are true, the edge to be traversed is non deterministically chosen. They deal strictly with control flow. There are no mapping rules for pins. Also, they do not deal with send signal or accept signal actions. They deal with the beginning of the execution of an activity diagram as an internal choice between initial nodes, but they do not take into account call behaviour action without incoming edges. They do not treat different kinds of action, for instance, call operation, call behaviour, and value specification. Other features of activity diagram, like accept event action, are translated into CSP.

Abdelhalim et al. \[A^{+10}\] propose the use of a subset of fUML (Foundational Subset for Executable UML) that is mapped into CSP \[\text{Hoa85}\]. Their focus is on analysis of dynamic behaviours. As control flow has been addressed by Xu et al. \[X^{+08}, X{\text{M}}^{09}\], they concentrate on mapping \textit{SendSignalAction} and \textit{AcceptEventAction} and signals \[A^{+10}\]. They deal with decision node as an internal choice. Also, they map expansion region into CSP processes. They deal with
signals by means of an asynchronous buffer, whereas in our translation we use a one-place synchronised buffer that could receive data from an asynchronous buffer. Their communication model allows storing of signals.

Bisztray, Ehrig, and Heckel [BEH07] define translation rules that relate edges in an activity diagram to a process in CSP [Hoa85]. They do not deal with object nodes or object flow, just with the translation of control flow. They translate a join node separately from the fork node. In other words, they have distinct translation rules for these control nodes. A consequence of that is that a synchronisation event appears only in the process that reach the join node, but not in the parallel operator that is introduced in the fork node. Also, the translation of the join node results in processes that are not similar: only one will behave as the process after the join, all the others will terminate in Skip. In our case, the reason for dealing with fork-join pairs is that if we treat them separately when we translate a join node it is necessary to change the CML part that corresponds to the fork node that has been translated before the join node.

Table 7.1 presents a comparison of the coverage of our formalisation with related work. The left column contains the features we formalised. The ✓ indicates that the feature is covered by related work (column), whereas × indicates it is not. We cover the following features: Call Behaviour Action (CBA), Interruptible Region (IRg), Call Operation Action (COA), Opaque Action (OpA), Accept Event Action (AEv), Send Signal Action (SSA), Value Specification Action (VSA), Initial Node (IN), Flow Final Node (FF), Decision Node (DN), Merge Node (MN), Fork Node (FN), Join Node (JN), Activity Final Node (AFN), Object Node (ObjN), Control Flow (CtrF), Object Flow (ObjF), and Activity Parameter Node (APN).
<table>
<thead>
<tr>
<th></th>
<th>X'08</th>
<th>XMP09</th>
<th>A'10</th>
<th>BEH07</th>
</tr>
</thead>
<tbody>
<tr>
<td>CBA</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>IRg</td>
<td>✓</td>
<td>✓</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>COA</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>OpA</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>AEv</td>
<td>×</td>
<td>✓</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>SSA</td>
<td>×</td>
<td>✓</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>VSA</td>
<td>×</td>
<td>✓</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>IN</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>FF</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>DN</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>MN</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>FN</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>JN</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>AFN</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>ObjN</td>
<td>×</td>
<td></td>
<td>×</td>
<td></td>
</tr>
<tr>
<td>CtrF</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>ObjF</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>APN</td>
<td>×</td>
<td>✓</td>
<td>×</td>
<td>×</td>
</tr>
</tbody>
</table>

Table 7.1: Coverage comparison
Chapter 8

Conclusions

In this chapter, we discuss and summarise our main contributions, and set an agenda for future work in task T2.2.2.

8.1 Summary

In this document we report on the formal link between SysML and CML by means of a denotational semantics. We have identified a restricted yet comprehensive set of diagrams and features that allows us to provide a semantics in CML.

We have formalised the semantics of blocks (as specified by block definition and internal block diagrams), state machine, sequence and activity diagrams. The semantics of blocks is central to the integration of both structural and behavioural aspects of complex SysML models into a single CML model. Whilst our semantics does not cover the totality of SysML, it provides a comprehensive treatment of the main types of diagrams that affect the behaviour of the system.

The semantics of blocks is given with respect to the underlying model of the block structure obtained by pre-processing the block definition and internal block diagrams. At the level of diagrams, different aspects of a block can be specified in different diagrams; the required pre-processing simply merges information that is spread across a number of diagrams to build an integral view of the elements of the model, and is carried out by the modelling tool when creating the model defined by the diagrams. Based on this integrated view, we propose translation rules for simple and composite blocks, as well as for ports. Blocks and ports are translated into CML processes, these processes may use other processes that model any associated state machines and activity diagrams. In the case of a composite block,
the process is defined in terms of the process that model the parts of the block. Associations between blocks (and ports) are modelled through alphabetised composition, where appropriate channel sets are defined to allow the communication between the blocks. In line with our guidelines, we support signals and operations, provided and required interfaces, and state machines as block behaviour specifications.

Our treatment of state machine diagrams covers a large number of elements, namely, simple, orthogonal and non-orthogonal states, regions, junctions, completion and non-completion transitions, join and fork pseudostates and final pseudostates.

Regarding sequence diagrams, although the main constructs have been translated, some other elements must be dealt in the future. The synchronization mechanism used for the message exchange can only be applied to the semantics of synchronous calls. Although we know some strategies to represent asynchronism in these cases, their adequacy will be further investigated. The remaining combined fragments not yet translated must be analysed to understand which ones are interesting for System of Systems modelling. Duration and time constraints also may have possible translation allowing strict restrictions on time. The use of state invariants may be one lead to integrate sequence and state machine diagrams. Another interesting possibility is to verify consistency of the relationships depicted in sequence diagram according to the ones modelled in structural diagrams and in the others behavioural diagrams. Finally, the use of allocations must be further investigated to check another possibilities for integration.

The translation rules for activity diagrams cover action nodes, object nodes, control nodes (decision, merge, fork, join, flow final, and activity final). Also, there are translations rules for Accept Event Action, and Send Signal Action. Communication is treated as synchronous, although Accept Event Action, for instance, involves asynchronous communication. We have translated object flows between object node also using synchronous communication.

To the best of our knowledge, no other work has formalised a semantics of SysML or UML models with the same level of coverage as the one presented in this report.

We observe main areas where development needs to be focused: validation of the models of SysML diagrams, and refinement in SysML. In the next sections, we briefly discuss each of them, identify the main challenges and discuss how these challenges may be overcome.
8.2 Validation

The models obtained through the translation strategies presented in this report have only been validated by manual translation, simulation and analysis of small examples. In order to simulate the models, the CML specifications are translated into CSP, and simulated and analysed using CSP tools: ProBE [For98] and FDR2 [For99]. The translation from CML to CSP is carried out manually, and we do not believe a general enough translation strategy between the two notations can be devised. Moreover, it can be difficult to interpret analysis and simulation results of the CSP models with respect to the CML models. This preliminary validation was made against our understanding of how the model should behave.

The next step in validating our models is to apply our translation rules in a more systematic and automatic fashion to support the generation of more complex diagrams. To this end, a prototype translation tool is being developed by Atego based on the translation rules for blocks, state machines and sequence diagrams. Furthermore, once the CML tools support simulation and analysis, we will be able to directly validate our semantics by both verifying properties of the models, and comparing simulation results of the CML models against the results of alternative simulation tools.

8.3 Refinement in SysML

Since CML is a refinement language, and our semantics of SysML is based on CML, it is only natural to think of refinement at the level of SysML diagrams. In fact, refinement has a two-fold role in our work:

1. Refinement provides the means for correct-by-construction simplifications of our models. The rules proposed in this document aim at providing a compositional and extendable semantics for SysML models, additionally we have avoided the inclusion of optimisations in our translation rules, for these reasons the models tend to be large and rely heavily on parallelism, which can lead to an increase in the cost of analysis as well as simulation. To circumvent these issues, refinement strategies can be devised to simplify and optimise the CML models, whilst preserving the original semantics. This approach presents an advantage over the optimisation of translation as such optimisation is necessarily manual and error prone, whilst models obtained through a refinement-based optimisation are correct by construction.

2. Refinement as a means for transforming SysML models. Particularly in the
field of software engineering, model transformation have been thoroughly investigated. However, these transformations are usually cast in a purely syntactic form, and semantic correctness is not formally verified. Refinement may be the key for supporting the development of formally verified model transformations.
Appendix A

Omitted translation rules

**Rule A.1** \(t_{\text{block\_signal\_type}}(b:\text{Block}): \text{class paragraph} =\)

```plaintext
public S = [for each s: b.Signals do
  name(s)
  end for]
if b.parent <> NULL then
  | name(b.parent) \_types ‘S
end if
;
```

**Rule A.2** \(t_{\text{block\_input\_type}}(b:\text{Block}): \text{class paragraph} =\)

```plaintext
public I = [for each op: b.Operations do
  name(op) \_I
  end for]
if b.parent <> NULL then
  | name(b.parent) \_types ‘I
end if
;
```
Rule A.3 \( t\_\text{block\_output\_type}(b: \text{Block}): \text{class\ paragraph\ } = \)

\[
\text{public } 0 = \text{[ for each } \text{op: } b.\text{Operations do ] } \\
\quad \text{name(op)}_0 \\
\text{end for ] } \\
\text{if } b.\text{parent } \not= \text{NULL then } \\
\quad \text{name(b.parent)}_0 \text{\_types}'0 \\
\text{end if ] } \\
\]

Rule A.4 \( \text{RUN}(S: \text{set of channel}) = \text{action\ paragraph\ } = \)

\[
\text{mu } x \oplus \{ \text{[ } \text{ev } \rightarrow x ] \mid \text{ev in } S \} \\
\]
Bibliography


[CHS00] K. Compton, J. K. Huggins, and W. Shen. A semantic model for the state machine in the unified modelling language. In Dy-


