Definition of the Formal Semantics of Control State Diagrams and Implementation of a Graphical Editor

Markus Leitz

Ulm University

Faculty of Engineering
Computer Science and Psychology

Institute of Software Engineering and Programming Languages

March 2018

Masterthesis in Software Engineering
Abstract

Abstract State Machines (ASM) are a formal specification language which is used in the software and system engineering. Different to UML2 diagrams, which are another well-known, graphical, formal specification language, ASMs are mostly used for scientific work and are textual based. Control State ASMs are a special class of ASMs which are similar to Finite State Machines, but extends them by parallelism and data structure manipulation.

To give a better overview of their Control State ASM specifications, flowchart-like charts are often made. They also help the reader understand the ASMs. Such graphical representation are then called Control State Diagrams (CSD). Because these CSDs are yet not fully specified, they often lead to confusion and misunderstandings, because often nothing is said about the behavior of CSDs and there is no full list of possible shapes of them.

This work will take the actual literature, analyzes the existing CSDs and the corresponding ASMs and derive a full definition of the syntax and semantics of CSDs in the form of a formal specification. Additionally a prototype of a graphical editor is implemented, with which CSDs can be created and transformed to CoreASM, a special ASM tool which is used to run ASMs on a computer.

Supervisor: Prof. Dr. Matthias Tichy
Supervisor: Prof. Dr. Egon Börger
Advisor: Dr. Alexander Raschke

Version March 15, 2018
Typeset: PDF-LATEX 2ε
## Contents

1 Introduction  
1.1 Motivation ........................................ 1  
1.2 Related Work ....................................... 7  
1.3 Fundamentals ....................................... 7  
1.3.1 Abstract State Machines ......................... 8  
1.3.2 Specification Definition ......................... 10  
1.4 Structure of the Work .............................. 12  

2 Abstract and Concrete Syntax ......................... 13  
2.1 Description ....................................... 13  
2.2 Structural Aspects ................................ 15  
2.3 Abstract Syntax .................................. 15  
2.4 Concrete syntax .................................. 22  

3 Semantics ............................................. 27  
3.1 Predefinitions and Notational Explanations ........ 27  
3.2 Translator ASM ................................... 31  
3.3 Mode translations ................................ 33  
3.4 Rule translations ................................ 36  
3.5 Conditional translations ......................... 40  

4 Extension: Substitution .............................. 47  
4.1 Introduction ..................................... 47  
4.2 Substitution specification ......................... 51  

5 Evaluation ............................................ 59  
5.1 Proof of completeness and consistency .......... 59  
5.2 Implementation ................................... 63
1 Introduction

Abstract State Machines (ASMs) are a formal system engineering method which guides the developer seamlessly from requirements capture to their implementation. Although ASMs have a mathematical foundation, a developer can correctly understand them as a pseudo-code without any special mathematical knowledge. As a formal method ASMs can be used to specify the requirements in a rigorous way and to later verify the software by reasoning techniques and validate it through simulation and testing. Over the years, different types and extensions of ASMs have been developed, which are used in different scenarios (see Section 1.3.1).

In general, textual formal methods like ASMs are often "too hard to understand and use in practice"[1], because they are often just big and complex texts, so they are not so common in industrial projects. An approach to make such textual formal methods more understandable, is to give a visualization of the text. This often results in graphical formal methods, like UML, which more often used in the industry [1].

Although ASMs can be understood by system- and software engineers, as mentioned above, a graphical representation is often used to support the understanding of an ASM. These diagrams are often oriented to state machines and other UML diagram types.

Because these diagrams are not well defined yet, which results in a couple of problems (see Section 1.1), the goal of this thesis is to give a specification of these diagrams. Therefore this section contains the motivation, why this specification is needed, followed by a conclusion of the related work and some fundamentals which give some basic knowledge about the important topics. At the end of this section, the structure and the scientific approach of this work will be explained.

1.1 Motivation

As described above, most system- and software developers can read ASMs without any deeper knowledge about ASMs. They can read it as a kind of pseudo-code and will understand the most of it right. The following TrafficLight ASM shows such a (simple) code, which describes the behavior of a single traffic light.
TrafficLight =

if phase = Stop then
  seq
    phase := GetReadyToGo
    SwitchLights(phase)
    Wait(2)
if phase = GetReady then
  seq
    phase := Go
    SwitchLights(phase)
    Wait(20)
if phase = Go then
  seq
    phase := GetReadyToStop
    SwitchLights(phase)
    Wait(2)
if phase = GetReadyToStop then
  seq
    phase := Stop
    SwitchLights(phase)
    Wait(20)
where SwitchLights(phase) =

if phase = Stop then
  YellowLight := off
  RedLight := on
if phase = ReadyToGo then
  YellowLight := on
if phase = Go then
  GreenLight := on
  YellowLight := off
  RedLight := off
if phase = ReadyToStop then
  GreenLight := off
  YellowLight := on
Often, ASM developer additionally use a special kind of diagram to give also a graphical representation beside the textual code. These diagrams are often orientated to flowchart-like diagrams like state charts or diagrams in BPMN notation. An examples of such diagrams is shown in Figure 1.1, where the functionality of an ATM\(^1\) is described. This diagram just uses ellipses and rectangles. Figure 1.2 from the same book shows an example of a forall rule, including hexagons and rectangles. The last example, shown in Figure 1.3 describes the knowledge transfer when people learn from each other. This diagram contains ellipses, rectangles and two kind of rhombus, which are like the shapes of the BPMN notation.

\(^1\)Automated Teller Machine
CHAPTER 1. INTRODUCTION

Figure 1.3: Example CSD with an other syntax which is similar to the BPMN notation [3, Fig. 2].

The problem with all of these diagrams is, that there is actually not any full specification of these diagrams, as they are not or not well introduced and described. In most cases, the diagrams are just printed beside the corresponding code while nothing is mentioned about the semantics or the full syntax [3]. In rare cases it is briefly explained by an example that such diagrams exist and how they are to be understood, like in [2]. Here these diagrams are just introduced by a short abstract example with the corresponding translation (see Figure 1.4) and the following short abstract introduction:

"For the graphical representation of control states we will use in this book both their inscription into circles [...] and the usual flowchart or UML notation where
the control states appear as named directed arcs (arrows) or as unnamed arcs. The former notation, which is common in automata theory, helps to visually distinguish the role of control states - to "pass control" - from that of ASM rules, which describe the update "actions" concerning the underlying data structure and are inscribed into rectangles, often separated from the rule guards which are written into rhombs or hexagons labeling the arcs outgoing the control states [...] or ingoing [...], following the practice of UML activity diagrams. The most common cases are those of diagrams with one or with two opposite conditions in the rhombs or hexagons ($n = 1, 2$); in the latter case usually the condition is written into the rhomb and the two exiting arcs are labeled with "yes" and "no", respectively [...], which are sometimes colored in grey to let them stand out better. [...] When using graphical notation we allow ourselves sometimes some self-explaining variations of the layout, which can always be reduced to the official definition explained above.”[2, pp. 45 – 46]

Figure 1.4: Introduction of control state diagrams in [2, p. 45, Fig. 2.5].

On closer inspection such a (too) short / inaccurate definition leads to further problems. For example as it is not described in detail what the meaning of a transition from one node to its successor means, the reader had to derive the meaning for himself by the given examples. This can lead to confusion or misunderstandings, depending on the reader’s prior knowledge. Figure 1.5 shows an alternative introduction from the same book compared to the first introduction, now a rule has been executed before the conditions $cond_1, \ldots, cond_n$ are checked. A reader with some experience with state charts or other flowchart-like diagrams may think the rule in this example is executed sequentially before the parallel execution of the condition statements. The ASM code right of the diagram says in difference that all statements, the conditions AND the rule are executed in parallel.

Another problem, which can be traced back to the central issue of the missing specification, is that the diagrams are not consistent. An example of this inconsistency can be found in the comparison of Figure 1.1 and Figure 1.3. Figure 1.1 just uses ellipses and rectangles while Figure 1.3 contains also shapes from the BPMN notation and generally uses more different shapes.
for different kind of rules. So not just the concrete syntax is different, but also the semantics are different. For other kind of diagrams, like state charts, which are better specified then CSDs, also different kind of representation exits, but they all are based on the same abstract model and so the semantics are equivalent and a reader can easily understand different kind of state charts.

Also striking is that without a precise definition, often even within a work different representations are used. An example of this can be found again in [2], where at the beginning the representation for \textit{forall} rules was introduced as a two-line rectangle, as shown in Figure 1.6, but was later used quite differently as a rectangle with a superimposed rectangle as the header, as shown in Figure 1.2.

\begin{center}
\begin{tabular}{|c|}
\hline
forall $x$ with $\varphi$ \\
$R$ \\
\hline
\end{tabular}
\hspace{1cm}
\begin{tabular}{|c|}
\hline
choose $x$ with $\varphi$ \\
$R$ \\
\hline
\end{tabular}
\end{center}

Figure 1.6: Suggested CSD construct for a \textit{forall} and a \textit{choose} rule [2].

Figure 1.3 and 1.2 also show another problem that occurs because of the missing or abstract specification, mostly not all shapes are explained but they are simply used in the diagrams. In Figure 1.3 e.g. nothing is said about the diagrams and the meaning of the rhombus with a circle inside is completely unclear. The symbol itself is known i.a. from the BPMN notation, but there it has the meaning of an Inclusion/OR split. Looking at the associated ASM code, it becomes clear that this is a \textit{choose} construct, which is more an XOR relative to BPMN.

In Figure 1.2 the meaning of the empty hexagon is not clear. Presumably this is a split/merge node which results in an infinite loop.

In order to solve all these problems and to avoid then in the future, a complete specification of the graphic representation which is developed in this thesis with the help of a model based approach is needed. This can then be used for future work to consistently produce diagrams that are easy to understand for anyone who knows the specification.

\textsuperscript{2}the concrete representation, see Section 1.3.2
1.2 Related Work

As mentioned earlier, there is currently no complete formal specification of Control State Diagrams. Arcaini et. al. describes how ASMs can be visualized and how both from an existing ASM codes a diagram and from a diagram ASM code can be generated [1]. The difference to this thesis is, that Arcaini et. al. do not give a formal specification of the graphical representation of ASMs, but only define a bi-directional translator by describing a couple of patterns of abstract ASM code and abstract diagram parts. The missing structural aspects, which are parts of this work, makes it difficult to understand whether the translator is complete or which ASM constructs are covered and how they relate to each other. This paper was used to get a first idea, how the semantics of CSDs can be described and also shows that this is not enough for a full formal specification.

The approach used to define the specification is inspired by the work of Kohlmeyer [4] and Sarstedt [5]. These works specified the dynamic behavior of different UML 2 diagrams using ASMs. They show what is needed for a good specification of languages and why ASMs are good for such specifications. Farahbod et. al. give another approach and notation to specify a language with ASMs [6]. Both approaches are combined in this thesis to define the specification of CSDs and fulfill both, an easy to understand pattern based translator ASM and a full definition of the structural aspects of CSDs by ASMs.

The basics of the topics ASM, Control State ASMs and their previous graphical representation are mainly derived from [2], [7], [8], [9] and [3], which are examples of work on ASMs in general and certain parts of ASMs that are important to this work, such as the refinement method, the use of ASMs in software development or the expansion of ASMs.

1.3 Fundamentals

As the previous introduction explains, this thesis is about Abstract State Machines (ASMs) with a closer look on the subclass of Control State ASMs and their graphical representation. To develop the specification of this graphical representation a model based approach is used. The following section give some fundamental information and definitions to these topics, which may help the reader to understand the upcoming sections.
1.3.1 Abstract State Machines

Abstract State Machines (ASM) are state machines which operate on states with an arbitrary data structure and are used as formal specification language in the system and software engineering. The first ideas and theories were made by Yuri Gurevich in the mid-1980s [10] and were later known as evolving algebra. As the name implies, these initial ideas were based on applied mathematical principles. During the next years Gurevich’s theory was enhanced to an applicable tool for the formal system and software specification. Especially the development of the ASM method by Egon Börger in the early 1990s [11] made ASMs practical [9].

During the development of ASMs and the ASM method it was often recognized that something is missing in the definition of the ASMs to make it practical for special use cases in the modern IT world. This has resulted in various extensions of ASMs to address the various problems that have surfaced in this evolution [2]. As an example, Börger and Schmidt extended the definition by concepts for the sequential execution, iteration and parameterization [12], which later results in the definition of TurboASMs [13].

For the better understanding of the functionality of ASMs, these are now briefly formally defined.

An ASM is formally defined by a signature \( \Sigma \), a set of initial states for \( \Sigma \), a set of rule declarations and a nullary function called main rule.

A signature \( \Sigma \) is a finite collection of functions. Each function has an arity, which is the number of arguments that the function takes. A nullary function is also called constant. A domain is a set of constant functions of a signature. An example for a signature is \( \Sigma_{\text{Bool}} = 0, 1, -, +, \ast \) where 0,1 are constants, - is a 1-ary function, which mean that it takes one argument and +,\ast are binary functions which takes two arguments. The corresponding domain Bool contains the elements 0 and 1.

A state is a non-empty set \( X \) (also known as the superuniverse) of the interpretation of the functions of a signature \( \Sigma \).

The dynamic behavior of an ASM is described by so called runs. A run contains a sequence of moves with each move contains an consistent update set.

An consistent update set is, as the name already says, a set of consistent (not clashing) updates. An update is a pair \((v, l)\) which describes for a given location \(l\) a new value \(v\). In a consistent update set, each update either do not contain the location of another update (not even just a
1.3. FUNDAMENTALS

<table>
<thead>
<tr>
<th>Rule Name</th>
<th>Notation</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Skip Rule</td>
<td>skip</td>
<td>Do nothing</td>
</tr>
<tr>
<td>Update Rule</td>
<td>f(s₁,…,sₙ) := t</td>
<td>Update the value of f at (s₁,…sₙ) to t</td>
</tr>
<tr>
<td>Block Rule</td>
<td>P par Q</td>
<td>P and Q are executed in parallel</td>
</tr>
<tr>
<td>Conditional Rule</td>
<td>if φ then P else Q</td>
<td>If φ is true, then execute P, otherwise execute Q</td>
</tr>
<tr>
<td>Let Rule</td>
<td>let x = t in P</td>
<td>Assign the value of t to x and then execute P</td>
</tr>
<tr>
<td>Forall Rule</td>
<td>forall x with φ do P</td>
<td>Execute P in parallel for each x satisfying φ</td>
</tr>
<tr>
<td>Choose Rule</td>
<td>choose x with φ do P</td>
<td>Choose an x satisfying φ and then execute P</td>
</tr>
<tr>
<td>Sequence Rule</td>
<td>P seq Q</td>
<td>P and Q are executed sequentially, first P and then Q</td>
</tr>
<tr>
<td>Call Rule</td>
<td>r(t₁,…,tₙ)</td>
<td>Call transition rule r with parameters t₁,…,tₙ</td>
</tr>
</tbody>
</table>

Table 1.1: Basic ASM rules [2]

part), or if two updates contain the same (part of a) update, the value of this (partial) location is in both updates the same.

A location \((f^n,(a_1…a_n))\) contains a n-ary function \(f\) and n elements \(a_1…a_n \in \text{State}\). The elements \(a_1…a_n\) are the arguments of the function \(f\) and describe a "location" in the state where \(f\) take place.

To orchestrate the updates within an update set, some special rule constructs, which are well known in the programming world, are used. Table 1.1 contains the basic rules for ASMs.

A special behavior of ASMs is, that the single updates within an update set are executed in parallel, only in case that updates are inside a sequence rule, they are executed sequentially.

Even though ASMs are industry-ready, they are still used primarily in science and research today. As a result, there are just rarely examples from commercial projects [2]. In science, for example, they found Use in the specification and verification of various programming languages such as Java[14] or Prolog [15].

Control State ASM

A special subclass of the basic ASMs are Control State ASMs. These frequently used kind of ASMs are similar to Finite State Machines (FSM) but extent them by parallelism and data structure manipulation [2]. The signature of a Control State ASM contains a special domain, which has the same meaning as a state in a FSM. In this thesis, this domain is named mode, in other books or papers often terms like state [3] or \(ctl_{state}\) [2] are used.
The rules of Control State ASMs are all of the form

\[
\text{if } mode = i \text{ then} \\
space \text{actions} \\
mode := j
\]

where \(i, j \in Mode\) and \(\text{actions}\) are rules like described in the ASM introduction part 1.3.1.

Often, however, a Control State ASM only consists of one rule which consists of many such condition rules. The general schema of such global rule is:

\[
\text{CONTROLSTATERule =} \\
\text{if } mode = m_i \text{ then} \\
\text{actions} \\
mode := m_j \\
\text{if } mode = m_j \text{ then} \\
\text{actions} \\
mode := m_k \\
\ldots \\
\text{if } mode = m_k \text{ then} \\
\text{actions} \\
mode := m_n
\]

Because of this form of rules, Control State ASMs are often used to describe the dynamic behavior of the systems and software to be specified, because, as with state machines, the process can be determined using a specific variable.

1.3.2 Specification Definition

To define a specification of a language, as what CSDs can be seen, it is mostly divided into four aspects: structure, constraints, representation and behavior [16]. Figure 1.7 shows how these aspects are related to each other.
1.3. FUNDAMENTALS

Figure 1.7: Aspects of a computer language description [17, Fig. 1].

**Structure:** All static information about the constructs of a language and how they are related. E.g. a natural language consists of numbers, letters and punctuation marks.

**Constraints:** Additional information about the structure which restricts the general structure to an allowed structure according to additional logical constraints. E.g. that in a simple natural language two consecutive punctuation marks are not allowed.

**Representation:** Defines how instances of the language are represented to human beings. This can be the description of a graphical or textual concrete language syntax [17]. E.g. the known symbols of the Roman alphabet.

**Behavior:** Describes how the model is used. This aspect includes execution of the language as well as mappings. By mapping a relation between the language itself and another representation is understandable. E.g. the meaning of a set of letters (word).

Traditionally there are two different kind of notations: Defining a grammar or a meta model [17].

In a grammar the structural aspects of a language are often defined together with the representation aspects by rules written in EBNF notation. The constraint aspects are mostly coded into the EBNF rules. The last aspect, the behavior, is not much formalized for grammars. In most cases a plain text describes in natural language the behavior of the specified language.

In the meta model approach the language is seen as a model which is defined by a to be created meta model. Therefore it combines the structural and the constraint aspect into the so-called
abstract syntax, the representation aspect is handled by the concrete syntax and the behavior is
defined by the semantics.

These three parts are each defined by an appropriate (formal) language. The abstract syntax
is mostly formulated by a class diagram which is derived from the meta model. The constraint
aspects of the abstract syntax, which normally can not be mapped with a (static) class diagram,
are written using certain constraint languages, e.g. the Object Constraint Language (OCL). The
concrete syntax is in most cases just a mapping from a class in the class diagram to a corresponding
representation\(^3\). This mapping can be formulated with functions or textual descriptions. The
semantics are often defined by different types of operational semantics notations, like triple-graph-
grammars or other pre-post-condition notations, or translational semantics where functions are
used to describe how the model is changed when it is in a special situation (which comply the
precondition of the function) like in graph transformations [18], [19].

1.4 Structure of the Work

The goal of this thesis is to give a full formal specification of the semantics and syntax of the
graphical representation of Control State ASMs. Therefore a design science research method like
described in [20] was used. The work treats the five steps of a design science research in a slightly
different order than they were originally intended.

First, the awareness of the problem was handled in this section by the motivation which
based on the literature review which is represented by the related work. In the following Section 2
contains the abstract and concrete syntax definitions of CSDs which are together with the
semantics definition in Section 3 the artifact of this research. Section 4 shows by an example how
the specification can be extended. The evaluation step of a design science research is handled
in Section 5 where the artifact is evaluated by a proof and the description of an implementation
of a tool for the generation and translation of CSDs. The two remaining steps suggestion and
development are both discussed in Section 6. Here are the most important points and problems,
which occur during the work, discussed and how they are solved. The last part of this thesis,
Section 7 contains a short conclusion of the work and an outlook about possible future work.

\(^3\)the representation can be in general both, textual or graphical
2 Abstract and Concrete Syntax

The next two sections contain the formal specification of Control State Diagrams. This specification is divided into a syntax part, containing the definition of the abstract and a concrete syntax in this section, and the definition of the semantics in Section 3. So in this section, the final abstract and a concrete syntax are defined. First, for the better understanding a textual description is given (Section 2.1) from which a class diagram is derived (Section 2.2). This class diagram contains the structural aspects of CSDs and is used to give a short overview over the structure of CSDs.

The formal abstract syntax definition, which is specified by an ASM, is given in Section 2.3. This includes also some validation predicates, which represent constraint aspects, which are not representable over the structural aspects. The last part 2.4 of this section contains the definition of a derived concrete syntax.

This definitions are based on the experience of the existing graphical representations of Control State ASMs. The abstract syntax is also influenced by the information from the definition of the Control State ASMs.

2.1 Description

CSDs can be described as:

CSDs are directed graphs that contains different kind of nodes and edges. A node is connected to another node with an edge. In some cases edges can optional be labeled. The basic types of nodes are: modes, rules and conditions.

Modes contain a label with the state of the mode they represent. Modes represent the beginning and the end of a Control State ASM rule. The beginning mode can therefore be also the end mode and there can be more then one end mode within a Control State ASM rule. Modes can have multiple incoming edges and, depending on the type of target nodes, different numbers of outgoing edges. So there can only be multiple outgoing edges
to rules and conditional nodes. A single transition to a mode is also allowed. Every CSD has exactly one mode which is marked as the initial start mode.

Rules also contain a label with the content of the rule, dependent of the type of the rule. Actually there are seven known types of basic ASM rules: skip-rules, update-rules, call-rules, forall-rules, choose-rules, let-rules and parallel-rules. Skip-rules contain a string label with the constant content Skip. Update-rules contain a label with the variables to be update and the new values in a string representation. Call-rules contain a string label with the name and the arguments of the called rule. Parallel-rules contain a list of rules which are executed parallel. Forall-rules contain a label with the loop header and a rule which is executed inside the loop. Choose-rules contain a string with the condition and a rule which is executed with the chosen parameter. Let-rules contain a string with the arguments and a rule which represents the scope of the let rule.

The last three rule types, forall, choose and let, can contain instead of an single rule to execute an embedded CSD. Embedded CSDs are a subclass of CSDs with the difference that they have some more restrictions. So the modes inside an embedded CSD have no outgoing edges and the start of such an embedded CSD had not to be a mode, but can be any kind of node.

All rule nodes can have multiple incoming edges, or if they are marked as the start Node of an embedded CSD also no incoming edge, and like modes also multiple outgoing edges to other rule nodes or conditional nodes. A single outgoing edge to a mode is also allowed.

Conditions contains a string with the condition which is evaluated to chose which path to go. Condition nodes can also have multiple outgoing edges but at least one. If there are more than one outgoing edges, these edges can be labeled with yes or no\(^1\). Not-labeled outgoing edges have the meaning that the evaluation of the condition is true.

If there are multiple outgoing edges with the same label, the targets of these transitions are either rules or conditions, as it is with rules and nodes with multiple outgoing edges.

A CSD contains also an identifier, including a name and optional arguments.

\(^{1}\)Or equivalent labels which represent the result of the evaluation of the condition.
2.2 Structural Aspects

Based on the natural language description, a class diagram was derived which represents the meta model in a formal language. This diagram is showed in Figure 2.1 and is used to give a better overview over the structural aspects of CSDS.

2.3 Abstract Syntax

Based on the previous description and structural aspects, an ASM is specified which represents the main part of this thesis. This ASM contains the abstract and concrete syntax definition of CSDs and their semantics definition. First of all, this section contains the parts about the abstract syntax, which can be divided into the structural and the constraint aspects.

The structure part consists of the basic domains which are needed for CSDs and some functions on these domains. One of these domains is a standard domain for Strings

- **domain String** \(\) sets of characters to represent text

The other main domains are derived from the previously generated metamodel (see Section 2), and are equivalent to the leaves of the metamodel tree (Figure 2.1) and can be divided into a couple of node types and one edge type. Following the metamodel the node data types are:

- **domain CSD**
- **domain EmbeddedCSD**
- **domain Mode**
- **domain SkipRule**
- **domain UpdateRule**
- **domain CallRule**
- **domain ForallRule**
- **domain ChooseRule**
- **domain ParallelRule**
- **domain LetRule**
- **domain Condition**
To get the information of a node, like the label of a mode, some functions are needed. The result of these functions is either an other kind of node or a set of nodes or a string which is inserted in the translation process without any plausibility check. That means for example that the content of a condition node is passed to the translated text (or code) without to check if the content contains a legal condition statement.

The functions are defined as follows:

\[
\text{modeLabel} : \text{Mode} \rightarrow \text{String} \\
\text{parallelRules} : \text{ParallelRule} \rightarrow \text{Set of Rule} \\
\text{callLabel} : \text{CallRule} \rightarrow \text{String} \\
\text{chooseLabel} : \text{ChooseRule} \rightarrow \text{String} \\
\text{chooseBody} : \text{ChooseRule} \rightarrow \text{Rule} \cup \text{EmbeddedCSD} \\
\text{forLabel} : \text{ForallRule} \rightarrow \text{String} \\
\text{forBody} : \text{ForallRule} \rightarrow \text{Rule} \cup \text{EmbeddedCSD} \\
\text{conditionLabel} : \text{Condition} \rightarrow \text{String} \\
\text{updateLabel} : \text{UpdateRule} \rightarrow \text{String} \\
\text{letLabel} : \text{LetRule} \rightarrow \text{String}
\]

Function to get the label of the mode

Function to get the rules inside the Parallel-Rule

Function to get the name and the parameters of the to be called rule in one string

Function to get the condition which is used to determine with which parameter the following rule is executed

Function to get the content which is executed. This can either be a single rule or an embedded CSD

Function to get the condition which must held for the elements of a given set for which a rule is executed

Function to get the rule or the embedded CSD which is executed within the forall rule

Function to get the condition to evaluate which path to go

Function to get the name and the new value of the variable which should be updated

Function to get Parameters which should be used in the let rule
2.3. ABSTRACT SYNTAX

\( \text{letBody} : \text{LetRule} \rightarrow \text{Rule} \cup \text{EmbeddedCSD} \)

Function to get the rule or embedded CSD which represent the the scope of the let rule

For better handling of the node domains, they are grouped to domains with different abstraction levels. So all rule domains are grouped to the **domain** Rule, which is formally defined by:

- **domain** Rule = \( \text{SkipRule} \cup \text{UpdateRule} \cup \text{CallRule} \cup \text{ForallRule} \cup \text{ChooseRule} \cup \text{ParallelRule} \)

The next abstraction level contains rules, modes and conditions and represents almost all leafs on the node side of the metamodel. So it is defined as:

- **domain** Node = Rule \( \cup \) Condition \( \cup \) Mode

After all node domains are defined, the domain of the other branch of the metamodel is still missing:

- **domain** Edge

An Edge is a tuple of two nodes \((\text{Source}, \text{Target})\) and connects these two nodes. In some cases Edges had to be labeled for example to evaluate which path after a condition represents the positive and which the negative result of the condition, so one edges is labeled with \text{yes} and the other edges with \text{no}. To get this information a function exists which is defined as follows:

\( \text{edgeLabel} : \text{Edge} \rightarrow \text{String} \)

Function to get the label of a Edge

The function value of \( \text{edgeLabel} \) can be \text{yes}, \text{no} or \text{undef}.

As it is described in the meta model, CSDs and EmbeddedCSDs consists of Nodes and Edges. The following functions are defined to get the information about a CSD:

\( \text{nodes} : \text{CSD} \rightarrow \text{Set of Node} \)

Function to get the nodes inside a CSD

\( \text{initialMode} : \text{CSD} \rightarrow \text{Mode} \)

Function to get the mode which is marked as the \text{initialMode} of a CSD

\( \text{name} : \text{CSD} \rightarrow \text{String} \)

Function to get the name of a CSD
Because EmbeddedCSD is a subclass of CSD these functions are also applicable for EmbeddedCSDs.

The following function is only defined for the EmbeddedCSDs:

\[
\text{startNode} : \text{EmbeddedCSD} \rightarrow \text{Node}
\]

Function to get the node which is marked as the startNode of an embedded CSD.

Note that an EmbeddedCSD can start with any kind of node (but exactly one), while a CSD always start with a mode (also exactly one).

The next step is to include the other part of the abstract syntax, the constraint aspects, into the ASM. This is achieved with special validation functions which ensure that only valid CSDs of this specification are sufficient. These validation functions are grouped in the \text{ValidCSD} function and is defined by:

\[
\text{ValidCSD \textbf{iff}}
\]

\[
\text{ValidModesAndRules and ValidConditions and ValidSuccessors}
\]

\[
\text{and ValidEmbeddedCSDs and NoInvalidLoops and NoEmptyNodes}
\]

These six validate functions are now explained in more detail.

First of all \text{ValidModesAndRules} checks if all nodes except conditions have either at least one incoming edge or they are marked as initialMode of a CSD respectively as startNode of an embedded CSD and all outgoing edges are not labeled.

\[
\text{ValidModesAndRules \textbf{iff forall} \ n \in \text{Mode} \cup \text{Rule}
\]

\[
(|\text{Incoming}(n)| \geq 1 \ \textbf{or} \ n \in \text{StartNode})
\]

-- at least one incoming edge (except for StartNodes)

\[
\text{and forall} \ e \in \text{Outgoing}(n) \ \text{edgeType}(e) = \text{empty}
\]

-- outgoing edges (if any) are followed unconditionally

where

\[
\text{StartNode} =
\]

\[
\{ \text{startNode}(csd) \mid csd \in \text{EmbeddedCSD} \} \cup \{ \text{initialMode} \}
\]
The two functions $\text{Incmoing}(n)$ and $\text{Outgoing}(n)$ are help functions which are frequently used in the validation functions to get the incoming respectively outgoing edges of a given node $n$. The exact definition are:

\[
\text{Incoming}(n) = \{(s, n) \mid (s, n) \in \text{Edge}\}
\]

\[
\text{Outgoing}(n) = \{(n, t) \mid (n, t) \in \text{Edge}\}
\]

The $\text{ValidConditions}$ functions checks if the constraints for the condition nodes holds for all of these nodes. This means that they have at least one incoming edge unless they are marked as $\text{startNode}$ and that they have at least one outgoing edge.

$\text{ValidConditions \text{ iff } forall } n \in \text{Condition}$

\[
(|\text{Incoming}(n)| \geq 1 \text{ or } n \in \text{StartNode})
\]

\[\text{ -- at least one incoming edge (except for } \text{StartNodes} \text{)}\]

and \[|\text{Outgoing}(n)| \geq 1 \]

\[\text{ -- at least one outgoing edge}\]

The $\text{ValidSuccessors}$ functions is a first simple check to avoid the creation of CSDs which would result in an inconsistent update, when they would be translated to ASM code. Other inconsistent update optimizations are not implemented in this thesis, see Section 6 and 7 for further information.

$\text{ValidSuccessors \text{ iff } forall } n \in \text{Node}$

if \[n \in \text{Mode} \cup \text{Rule}\] then

if \[|\text{Outgoing}(n)| > 1\] then

\[\text{there is no } (n, t) \in \text{Edge with } t \in \text{Mode}\]

else

\[\text{ -- } n \text{ is condition node}\]

if \[|\text{OutgoingYes}(n)| > 1\] then

\[\text{there is no } (n, t) \in \text{OutgoingYes with } t \in \text{Mode}\]

and if \[|\text{OutgoingNo}(n)| > 1\] then

\[\text{there is no } (n, t) \in \text{OutgoingNo with } t \in \text{Mode}\]

where
CHAPTER 2. ABSTRACT AND CONCRETE SYNTAX

OutgoingYes(n) =
{ e | e ∈ Outgoing(n) and edgeType(e) ∈ {yes, empty} }

OutgoingNo(n) = { e | e ∈ Outgoing(n) and edgeType(e) = no }

The next function, ValidEmbeddedCSDs checks, if the constraints of EmbeddedCSDs are complied. This contains that an EmbeddedCSD has exactly one startNode, the graph is independent from the surrounding CSD\(^2\) and that it contains no modes.

ValidEmbeddedCSDs iff \forall csd ∈ EmbeddedCSD\n  startNode(csd) ≠ undef\n  and ClosedSubgraph((nodes(csd), edges(csd)), (Node, Edge))\n  and \forall node ∈ modes(csd) | |Outgoing(node)| = 0

where

EmbeddedCSD =
{ forallBody(node) | node ∈ ForallRule\n  and forallBody(node) ∈ CSD }\n∪{ chooseBody(node) | node ∈ ChooseRule\n  and chooseBody(node) ∈ CSD }\n∪{ letBody(node) | node ∈ LetRule and letBody(node) ∈ CSD }\nClosedSubgraph((Node', Edge'), (Node, Edge)) iff\n
Node' ⊆ Node and Edge' ⊆ Edge\nand thereisno (s, t) ∈ Edge' with\n  s ∈ Node' and t ∈ Node \setminus Node'\nand thereisno (s, t) ∈ Edge' with\n  s ∈ Node \setminus Node' and t ∈ Node'

\(^2\)There are no connections from the EmbeddedCSD to the surrounding CSD and vice versa
NoEmptyNodes checks if all nodes contains either a label, a Rule or an EmbeddedCSD.

\[ \text{NoEmptyNodes iff } \forall \text{ node } \in \text{Node} \] 
\[ \quad \text{-- some label is defined} \]
\[ \quad \text{if node } \in \text{Mode then modeLabel(node) } \neq \text{ undef} \]
\[ \quad \text{if node } \in \text{Condition then conditionalCondition(node) } \neq \text{ undef} \]
\[ \quad \text{if node } \in \text{CallRule then callLabel(node) } \neq \text{ undef} \]
\[ \quad \text{if node } \in \text{UpdateRule then updateLabel(node) } \neq \text{ undef} \]
\[ \quad \text{if node } \in \text{ParallelRule then parRules(node) } \neq \text{ undef} \]
\[ \quad \text{if node } \in \text{ForallRule then} \]
\[ \quad \quad \text{forLabel(node) } \neq \text{ undef and forBody } \neq \text{ undef} \]
\[ \quad \text{if node } \in \text{ChooseRule then} \]
\[ \quad \quad \text{chooseLabel(node) } \neq \text{ undef and chooseBody } \neq \text{ undef} \]
\[ \quad \text{if node } \in \text{LetRule then} \]
\[ \quad \quad \text{letLabel(node) } \neq \text{ undef and letBody } \neq \text{ undef} \]

The NoInvalidLoops validation is needed for the translation process. It says that between two states, no loop containing just rules and conditions are allowed. In other words, inside of a subgraph, which is bounded by states and do not contain other states, a topological ordering had to be able to be done, which means that this inner subgraph is a directed acyclic graph. The algorithm for this check is a simple depth-first-search algorithm.

\[ \text{NoInvalidLoops iff } \forall \text{ m } \in \text{Mode} \]
\[ \quad \text{Loopfree(m)} \]
\[ \quad \text{where} \]
\[ \quad \quad \text{Loopfree(m) iff } \forall \text{ n } \in \text{Rule } \cup \text{ Condition with } (m, n) \in \text{Edge} \]
\[ \quad \quad \quad \text{NoLoopPath(n, \{\}} \]
\[ \quad \quad \quad \text{NoLoopPath(n, p) iff n } \notin \text{ p} \]
\[ \quad \quad \quad \text{and } \forall \text{ n' } \in \text{Rule } \cup \text{ Condition with } (n, n') \in \text{Edge} \]
\[ \quad \quad \quad \quad \text{NoLoopPath(n', p \cup \{n\})} \]
CHAPTER 2. ABSTRACT AND CONCRETE SYNTAX

The algorithm checks for every part of the control state rule, which is bounded by two states, if the rules and conditions inside this part do not build a cycle. Therefore a set of nodes is created (which is initially empty) and for every successor rule or condition a function is called which takes the successor node and the set of nodes and first checks if the given node is in the set of nodes, this would mean that a cycle is detected and the CSD contains an invalid loop, and if not, the function is recursively called for all successor nodes and a copy of the set of nodes extended by the actual given node.

2.4 Concrete syntax

As second part of the ASM one kind of concrete syntax\(^3\) is introduced. The following graphical syntax is based on the most popular diagrams by Egon Börger [2] and is expended by the objects that doesn’t yet exists.

First of all, the representation that are adopted from [2]:

<table>
<thead>
<tr>
<th>Representation</th>
<th>Class name</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>(m)</td>
<td>Mode</td>
<td>A mode is represented as circle with an inner label. To get the content of the label, the function (modeLabel) can be used</td>
</tr>
<tr>
<td>(f(s_1, \ldots, s_n) := t)</td>
<td>UpdateRule</td>
<td>An update rule is represented as a rectangle with an inner label that contains the variable and the new value which should be updated. To get this label, the function (updateLabel) can be used</td>
</tr>
<tr>
<td>(r(t_1, \ldots, t_n))</td>
<td>CallRule</td>
<td>A call rule is represented as a rectangle with an inner label that contains the rule and the parameters which should be called. To get this label, the function (callLabel) can be used</td>
</tr>
</tbody>
</table>

\(^3\)The model based approach which is used in this specification definition says that there can be several types of concrete syntax which are all derived from the underlying abstract syntax.
### 2.4. CONCRETE SYNTAX

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \forall x \text{ with } \varphi )</td>
<td>ForallRule</td>
<td>A forall rule is represented as a rectangle with two rows divided by a solid line. The upper row contains information about the forall header and can received with the \textit{forLabel} function. The lower row contains the rule or the embedded CSD which should be executed, to get this content the function \textit{forBody} can be used.</td>
</tr>
<tr>
<td>( \text{choose } x \text{ with } \varphi )</td>
<td>ChooseRule</td>
<td>A choose rule is represented as a rectangle with two rows divided by a solid line. The upper row contains information about the choose header and can received with the \textit{chooseLabel} function. The lower row contains the rule or embedded CSD which should be executed, to get this content the function \textit{chooseBody} can be used.</td>
</tr>
<tr>
<td>( \varphi )</td>
<td>Condition</td>
<td>A condition is represented as a hexagon with an inner label. To get the content of the label, the function \textit{conditionLabel} can be used.</td>
</tr>
</tbody>
</table>

For the other kind of nodes, a new representation had to be developed. Care was taken to ensure consistency with the existing forms. For example the let rule is similar to the forall and choose rule as it has a kind of header and a scope where the header is active.

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textbf{Skip}</td>
<td>SkipRule</td>
<td>A skip rule is represented as a rectangle with an inner label that always contains \textbf{Skip}.</td>
</tr>
</tbody>
</table>
CHAPTER 2. ABSTRACT AND CONCRETE SYNTAX

A let rule is represented as a rectangle with two rows divided by a solid line. The upper row contains information about the let header and can received with the `letLabel` function. The lower row contains the rule or embedded CSD which should be executed, to get this content the function `letBody` can be used.

A parallel rule is represented as a rectangle with n rows divided by a solid line. Each row contains a rule and all rules are executed parallel. To get the list of these rules, the function `parallelRules` can be used.

The nodes where the `initialMode` or `startNode` attribute is set can be recognized by a wider border.

The edges are represented as follows:

- Depending on the type and number of source and target nodes, an edge has a different meaning.
2.4. CONCRETE SYNTAX

Figure 2.1: Metamodel CSD
3 Semantics

This section contains the third and last part of the specification, which is the definition of the semantics of CSDs. Therefore a translator ASM is created which works as a translator to get the corresponding ASM notation of an given CSD. This translator ASM uses an algorithm which iterates over the given CSD and sequentially looks at each node and searches for a matching pattern corresponding to the current node and its successors, and then inserts the appropriate translation into a translated text.

The first part of this section contains some predefinitions and also some functional and notational explanations, which may help the reader to understand the functionality of the translator machine. The second part of this section contains the translator ASM specification.

3.1 Predefinitions and Notational Explanations

To get a better understanding of the following definitions and explanations, first of all the translation algorithm will be briefly explained.

The translator uses an algorithm which runs multiple times. In the first run (and only in the first run), a InitializeTranslator function is called, which initializes all relevant domains and objects. After that, in every run of the machine, the algorithm looks for a node which was not visited yet and calls a Translate function which searches for a pattern that matches the chosen node and his successors and adds the corresponding translation to an ASM translation object. If no more node is active, the translation process is finished and the translator machine terminates.

The patterns that are searched for a match contain combinations of nodes with a single node and its successor nodes\(^1\). Patterns with multiple incoming edges could not be separately viewed because they have the same meaning as the set of single patterns with one source and one target node. Figure 3.1 shows by a schematic example how a pattern with multiple incoming edges can be replaced by multiple patterns without multiple incoming edges. This means a pattern with multiple outgoing edges is handled by the patterns with the single source node.

\(^1\)This means no node in such a pattern has multiple incoming edges
Figure 3.1: Example of how combination patterns with multiple incoming edges are handled. On
the left side is the schematic pattern with multiple incoming edges and on the right side the alternative schematic representation without multiple incoming edges. The translation is always the same.

To make the translation process simpler, two generalizations are made. First, because all kind of rule nodes and condition nodes have the same behavior as target node, they are all represented with a special abstract rule node. Such an abstract rule node is represented as a rectangle with a double lined border and the label $rc^2$ inside: $rc^2$.

A second generalization groups all combinations with multiple successor nodes of the same type (mode or abstract rule). Instead of handle each of this potentially infinity combinations, all combinations with such multiple successors are grouped and represented by by two successors of the corresponding type with three dots in between. This is valid because the concrete number of outgoing edges is not relevant to the meaning if there is more than one edge. Figure 3.2 shows such a pattern with one rule as source node and the representation of multiple abstract rules as target nodes. This pattern matches all combinations with a rule as source node and at least two nodes as target nodes, where each target node is either a rule or a condition.

![Figure 3.2: Example of a pattern with multiple outgoing edges. A rule node can have multiple outgoing edges to different nodes (rules or conditions) which are represented as abstract nodes to make the translation rules more generic.](image)

With this generalization all combination of nodes with just sources and targets can systematically be handled. This guarantees that all combinations are analyzed and because the translation is not influenced by the context of the combination the whole translation is complete and, under

---

2 Rules and Conditions
3 The context is defined by the nodes and the edges that occurs before and after the combination

28
the condition that the function values are different, consistent. For more information about
the completeness and the consistency see Section 5.1 and the discussion about the systematical
combination handling in Section 6.

To make the translation patterns easier to understand, they are not described by the names of
the nodes, but instead the graphical representations from the concrete syntax is used within the
translator ASM.

The Translate function of the translator ASM uses a special notation to describe a pattern
and which translation steps are executed if this pattern matches. Figure 3.3 shows an example
of such a part of the function. It contains the combination pattern with a rule as source node
and a mode as target node (left side). When this pattern matches, the action part on the right
side is executed. Such an action part contains some statements to execute, like the initialization
of variables and always the call of the function AddTextualASM which gets a position (see
Section 3.2 for more information about positions) and a translation object. The Translate
function, which is specified later, contains for every valid pattern such a part.

![Figure 3.3: Example of a translation rule with the graphical combination pattern of a rule to a
mode node on the left side and the corresponding translation actions in the right side.](image)

The pattern notation on the left side of the translation rule is given by a graphical representa-
tion which is just a short notation to describes the given pattern. For example the pattern
\[
\text{\begin{tikzpicture}
\node (r) at (0,0) {rule};
\node (m) at (1,0) {m};
\draw (r) -- (m);
\end{tikzpicture}}
\]
if node = rule and rule ∈ Rule and m ∈ Mode

and (rule, m) ∈ Edge then

action

Where action is the right side of a part of the translation function. In the example, this would be the initialization of a position variable α, the call of the AddTextualASM function with the given position (pos) and the corresponding translate object. In the example also a function TranslateRule is called, to translate the concrete rule and as last point, the status of the mode m is set to active, if it wasn’t closed before.

The translation object, contains the translation of the pattern which is inserted into the translation text. To differ it from the other statements of the action part, the translation object is framed with a solid line. Because the translation is a dynamic process, the Translate function need to evaluate some functions which return the concrete values of the translation object. For example the label of a mode is not always the same, so it must first be extracted from the given mode object by calling the modeLabel function. Such function calls are printed inside the translation object and can be identified by the cursive font style. Text parts written in monospace style can be directly inserted into the translation text. Figure 3.4 shows such a translation object from the combination pattern with a condition as source node and multiple abstract rules as target nodes for the positive evaluation of the condition, and a single mode for the negative evaluation. The Greek Letters α₁...αₙ are position markers. The concept of these position markers is explained in the next section.

![Figure 3.4: Example of a translation. Translated text is written in monospace font (3). Functions which must be first evaluated are written in italic font (1). α is the position marker where the next translation should be inserted (2).](image-url)
3.2 Translator ASM

Next, the translator ASM is introduced. Therefore first two domains and a couple of functions are needed. Remember: The translation algorithm takes in each iteration a node which was not visited before, search for the combination pattern, which matches to the node and the targets of his outgoing edges and generate the translation of this pattern. So to recognize which node was already visited, the node class gets a new property status. A node then can have the status undef, active or closed

- **domain** Status $= \text{Def} \{\text{undef, active, closed}\}$

Because a status is binded to a node, it can also be viewed as a function status : Node $\times$ Status which is set by \(\text{status}(m) := \text{active}\) and can be get by \(\text{status}(m)\).

To get the position where the translation should be inserted a corresponding domain Position$^4$ is introduced. While it should be possible to set multiple positions at once, each position is also binded to the translation it belongs to, represented by the target node of the translated combination. This is for example needed when a condition with two outgoing edges is translated. Then one position for the positive evaluation path is set and one for the negative evaluation path.

- **domain** Position describes a position in the translation

The function pos : Node $\rightarrow$ Position can be used to get a position and pos\((m) := \text{position}\) to set it.

By definition it is possible that a node can have multiple incoming edges. Figure 3.5 shows a scenario, where a rule rule\(_2\) has two incoming edges, one from the no-path from condition cond and one from the mode \(m_3\). In such cases, the node with the multiple incoming edges and all of his (transitive) successor nodes has to be translated for each incoming path until a mode node is reached. Therefore a function positions is needed, which contains all Positions that belongs to a node, so formally:

\[
\text{positions} : \text{Node} \rightarrow p(\text{Position})
\]

\(\text{Function to get the set of positions (p) with all positions of a Node.}\)

$^4$An abstract data type which represents a position in any context. This can be an offset in a text, or a node in an abstract syntax tree or something else. In ASMs there is no need to describe how such a data type can be later implemented.
Figure 3.5: Example of a CSD where a node (rule “rule2”) has two incoming edges. The translation algorithm has to remember two positions for this rule, one for the path $m_1 \rightarrow \text{cond} \rightarrow \text{rule2} \rightarrow m_1$ and one for the path $m_3 \rightarrow \text{rule2} \rightarrow m_1$. 

The translator ASM TranslateCSD, can now be specified by:

$$\text{TranslateCSD}(\text{csd}) =$$

if notInitialized $=$ true then

$\text{InitializeTranslator}(\text{csd})$

notInitialized $:= \text{false}$

if notInitialized $=$ false then

choose node $\in \text{nodes}(\text{csd})$ with status(node) $=$ active

if node $\in \text{Mode}$ then

$\text{Translate}(\text{node}, \text{nextModePos})$

status(node) $:= \text{closed}$

-- a mode node is immediately closed

else

choose pos $\in \text{positions}(\text{node})$

$\text{Translate}(\text{node}, \text{pos})$

$\text{Delete}(\text{pos}, \text{positions}(\text{node}))$

if positions(node) \{pos\} $=$ $\emptyset$ then

status(node) $:= \text{closed}$

-- if no positions left, node is closed

The InitializeTranslator creates a new object where the resulted translation is inserted. Because this result had not to be further defined for the formal specification, the type of this object
stays abstract and the function \texttt{CREATE\_TEXTUAL\_ASM}, which create and fill such an object, is not further specified. Additional the function sets a new nextModePos (which is introduced in Section 3.3) and the status of the initialMode to active.

\begin{verbatim}
INITIALIZE\_TRANSLATOR(csd) =
  let \( \beta \) = \textbf{new} (Position)
  CREATE\_TEXTUAL\_ASM(
    name(csd) = \beta
  )
  nextModePos := \( \beta \)
  status(initialMode(csd)) := active
\end{verbatim}

The \texttt{TRANSLATE} rule contains all combination patterns with the corresponding action. So the general scheme of this rule is:

\begin{verbatim}
TRANSLATE(node, pos) =
  \{ pattern\(_i\) (node) \} \Rightarrow actions\(_i\)(node, pos)
  ...
  \{ pattern\(_n\) (node) \} \Rightarrow actions\(_n\)(node, pos)
\end{verbatim}

Where \texttt{node} is the source node of the pattern and \texttt{pos} is the position where the translated text should be inserted.

In the following, all patterns and the actions to translate are introduced and shortly described. They are grouped by the source node of the pattern.

### 3.3 Mode translations

Patterns with a mode as source node are treated special as the given position is the nextModePos. The nextModePos describes the next conditional position of the control state rule. Remember, a control state rule consists of single conditions.
CONTROLSTATERule:

\[
\text{if } \text{mode} = m_1 \text{ then }
\]

Part \(m_1\)

;

\[
\text{if } \text{mode} = m_n \text{ then }
\]

Part \(m_n\)

So nextModePos points to the position where the next part

\[
\text{if } \text{mode} = m_{n+1} \text{ then }
\]

Part \(m_{n+1}\)

is inserted. After each of this translations, the nextModePos had to be set to the position where the next control state rule part should be inserted.

In the following are all patterns with a mode as source node which are translatable by definition and their corresponding action.

\[
\left( \begin{array}{c}
\text{m}_1 \\
\text{m}_2 
\end{array} \right) \Rightarrow
\begin{aligned}
&\text{let } \beta = \text{new} (\text{Position}) \text{ in }

&\text{AddTextualASM}(\text{pos,}

&\begin{array}{l}
\text{if } \text{mode} = \text{modeLabel}(m_1) \text{ then } \\
\text{mode} := \text{modeLabel}(m_2)
\end{array}

&)

&\text{if } m_1 \neq m_2 \text{ and status}(m_2) \neq \text{closed} \text{ then }

&\begin{array}{l}
\text{status}(m_2) := \text{active}
\end{array}

&\text{nextModePos} := \beta
\end{aligned}
\]
A special case are modes without an outgoing edge. By definition this kind of mode is valid.
The meaning of such a pattern is that the ASM end here and will just skip.
3.4 Rule translations

Patterns with a rule as source node are similar to patterns with a mode as source node. The difference is that the translation text is not inserted at the nextModePos. Another difference is that the rule, which can be any kind of rule node, is translated separately in the TRANSLATERULE rule. This function takes a node and a position and works like the TRANSLATE function with the pattern matching.
3.4. RULE TRANSLATIONS

\[
\text{TranslateRule}(\text{node, pos}) =
\]

\[
\begin{align*}
\left( \begin{array}{c}
\text{Call}(x) \\
\end{array} \right) & \Rightarrow \text{AddTextualASM}(\text{pos}, \text{callLabel}(	ext{node})) \\
\left( \begin{array}{c}
x := \text{updateValue} \\
\end{array} \right) & \Rightarrow \text{AddTextualASM}(\text{pos}, \text{updateLabel}(	ext{node})) \\
\left( \begin{array}{c}
\text{skip} \\
\end{array} \right) & \Rightarrow \text{AddTextualASM}(\text{pos}, \text{skip}) \\
\end{align*}
\]

\[
\begin{align*}
\left( \begin{array}{c}
\text{RULe}_1 \\
\vdots \\
\text{RULe}_n \\
\end{array} \right) & \Rightarrow \text{let } \alpha_1 = \text{new (Position), } \\
& \quad \ldots, \\
& \quad \alpha_n = \text{new (Position) in } \\
& \quad \text{AddTextualASM}(\text{pos}, \text{par} \alpha_1 \ldots \alpha_n \text{ endpar}) \\
& \quad \text{forall } n_i \in \text{parallelRules(node)} \\
& \quad \text{TranslateRule}(n_i, \alpha_i)
\end{align*}
\]

37
\( \forall x \text{ with } \varphi \)  

\[
\Rightarrow \quad \text{let } \alpha = \text{new} (\text{Position}) \text{ in } \\
\text{AddTextualASM}(\text{pos}, \text{forLabel}(\text{node}) \text{ do } \alpha) \\
\text{if } \text{forBody}(\text{node}) \in \text{Rule} \text{ then } \\
\text{TRANSLATERULE}(\text{forBody}(\text{node}), \alpha) \\
\text{else} \\
\text{status}(\text{startNode}(\text{forBody}(\text{node}))) := \text{active} \\
\text{INSERT}(\alpha, \text{positions}(\text{startNode}(\text{forBody}(\text{node}))))
\]

\( \text{choose } x \text{ with } \varphi \)  

\[
\Rightarrow \quad \text{let } \alpha = \text{new} (\text{Position}) \text{ in } \\
\text{AddTextualASM}(\text{pos}, \text{chooseLabel}(\text{node}) \text{ in } \alpha) \\
\text{if } \text{chooseBody}(\text{node}) \in \text{Rule} \text{ then } \\
\text{TRANSLATERULE}(\text{chooseBody}(\text{node}), \alpha) \\
\text{else} \\
\text{status}(\text{startNode}(\text{chooseBody}(\text{node}))) := \text{active} \\
\text{INSERT}(\alpha, \text{positions}(\text{startNode}(\text{chooseBody}(\text{node}))))
\]

\( \text{let } x = \text{VALUE} \)  

\[
\Rightarrow \quad \text{let } \alpha = \text{new} (\text{Position}) \text{ in } \\
\text{AddTextualASM}(\text{pos}, \text{letLabel}(\text{node}) \text{ in } \alpha) \\
\text{if } \text{letBody}(\text{node}) \in \text{Rule} \text{ then } \\
\text{TRANSLATERULE}(\text{letBody}(\text{node}), \alpha) \\
\text{else} \\
\text{status}(\text{startNode}(\text{letBody}(\text{node}))) := \text{active} \\
\text{INSERT}(\alpha, \text{positions}(\text{startNode}(\text{letBody}(\text{node}))))
\]
The valid pattern for combinations with a rule as source node are:

\[
\left( \begin{array}{c}
\text{rule} \\
\text{m}
\end{array} \right) \Rightarrow \text{let } \alpha = \text{new (Position)} \quad \text{AddTextualASM(pos,} \\
\quad \alpha \\
\quad \text{mode := modeLabel(m)} \quad \text{)} \\
\quad \text{TranslateRule(rule, } \alpha) \\
\quad \text{if status(m) } \neq \text{ closed then} \\
\quad \quad \text{status(m) := active} \\
\quad \text{let } \alpha_1 = \text{new (Position),} \\
\quad \quad \ldots, \\
\quad \text{let } \alpha_n = \text{new (Position),} \\
\quad \alpha_{\text{rule}} = \text{new (Position)} \\
\quad \text{AddTextualASM(pos,} \\
\quad \quad \text{seq} \\
\quad \quad \quad \alpha_{\text{rule}} \\
\quad \quad \quad \text{par} \\
\quad \quad \quad \quad \alpha_1 \\
\quad \quad \quad \quad \ldots \\
\quad \quad \quad \quad \alpha_n \\
\quad \quad \quad \text{endpar} \\
\quad \quad \text{endseq} \quad \text{)} \\
\quad \text{TranslateRule}([\text{rule, } \alpha_{\text{rule}}] \quad \\
\quad \text{forall } n_i \in \text{Successors(rule)} \\
\quad \quad \text{status}(n_i) := \text{active} \\
\quad \text{INSERT}(\alpha_i, \text{positions}(n_i)) \quad)
\]
A single rule node is, like it was at the mode based combinations, a special case. By definition this kind of rule is valid. The meaning of such a pattern is that the part of the ASM ends here.

\[
\begin{align*}
\text{rule} \Rightarrow \text{TRANSLATERULE}(\text{rule}, \text{pos})
\end{align*}
\]

Also like mode based combination patterns, combinations with multiple outgoing edges with at least one mode as target node are not valid (see Fig. 3.7).

Figure 3.7: Not valid patterns with a rule as source node. This pattern is not valid because if a rule has an outgoing edge to a mode, this should be the only outgoing edge.

3.5 Conditional translations

Patterns of combinations with a condition as source node are different to mode and rule based pattern combinations as they allow in certain circumstances more then one outgoing edges to modes and that sometimes the edges had to be labeled. The following patterns contain all valid combination based on conditions.

\[
\begin{align*}
\text{cond} \to m \Rightarrow \text{ADDTXTUALASM}(\text{pos}, \\
\begin{cases}
\text{if} \ \text{conditionLabel}(\text{cond}) \ \text{then} \\
\quad \text{mode} := \text{modeLabel}(m)
\end{cases} \\
\text{if} \ \text{status}(m) \neq \text{closed} \ \text{then} \\
\quad \text{status}(m) := \text{active}
\end{align*}
\]
3.5. CONDITIONAL TRANSLATIONS

This is the only valid case with more than one mode as a successor to a condition. But this is only possible if one of the outgoing edges is labeled with "yes" and the other with "no".

```
cond yes m1
no m2

AddTextualASM(pos,

if conditionLabel(cond) then
  mode := modeLabel(m1)
else
  mode := modeLabel(m2)
)

if status(m1) ≠ closed then
  status(m1) := active
if status(m2) ≠ closed then
  status(m2) := active
```

```
let α1 = new (Position),
  ..., αn = new (Position),

AddTextualASM(pos,

if conditionLabel(cond) then
  par
    α1
  ...
  endpar
)

forall ni ∈ Successors(cond)
  status(ni) := active
  INSERT(αi, positions(ni))
```
\[ \begin{align*}
\text{let } \alpha_1 &= \text{new} \ (\text{Position}), \\
\ldots, \\
\alpha_n &= \text{new} \ (\text{Position}), \\
\text{AddTextualASM}(pos, \\
\quad \text{if } \text{conditionLabel}(cond) \text{ then} \\
\quad \phantom{\text{if }} \text{par} \\
\quad \phantom{\text{par }} \alpha_1 \\
\quad \phantom{\text{par }} \ldots \\
\quad \phantom{\text{par }} \ldots \\
\quad \phantom{\text{par }} \alpha_n \\
\quad \phantom{\text{par }} \text{endpar} \\
\text{else} \\
\phantom{\text{else}} \text{mode} := \text{modeLabel}(m) \\
\end{align*} \]

\[
\begin{align*}
\text{if } \text{status}(m) \neq \text{closed} \text{ then} \\
\phantom{\text{if }} \text{status}(m) := \text{active} \\
\end{align*} \]

\[
\begin{align*}
\text{forall } n_i \in \text{Successors}(cond) \\
\phantom{\text{forall }} \text{status}(n_i) := \text{active} \\
\end{align*} \]

\[
\begin{align*}
\text{INSERT}(\alpha_i, \text{positions}(n_i))
\end{align*} \]
3.5. CONDITIONAL TRANSLATIONS

\[
\text{let } \alpha_1 = \text{new} (\text{Position}), \\
\ldots, \\
\alpha_n = \text{new} (\text{Position}), \\
\text{AddTextualASM}(pos, \\
\begin{align*}
\text{if } & \text{conditionLabel}(\text{cond}) \text{ then} \\
& \text{mode } := \text{modeLabel}(m) \\
\text{else} \\
& \text{par} \\
& \alpha_1 \\
& \ldots \\
& \ldots \\
& \ldots \\
& \alpha_n \\
& \text{endpar}
\end{align*}
\]

\[
\text{if } \text{status}(m) \neq \text{closed} \text{ then} \\
\text{status}(m) := \text{active} \\
\text{forall } n_i \in \text{Successors}(\text{cond}) \\
\text{status}(n_i) := \text{active} \\
\text{INSERT}(\alpha_i, \text{positions}(n_i))
\]
let $\alpha_1 = \text{new} (\text{Position}),$

$\ldots,$

$\alpha_n = \text{new} (\text{Position}),$

$\beta_1 = \text{new} (\text{Position}),$

$\ldots,$

$\beta_m = \text{new} (\text{Position})$

\[ \text{ADDTEXTUALASM}(pos, \text{conditionLabel}(\text{cond}), \text{par} \begin{align*} \alpha_1 \cdot \\ \cdot \\ \cdot \\ \alpha_n \end{align*} \text{endpar} \text{if} \text{conditionLabel}(\text{cond}) \text{then} \text{par} \begin{align*} \beta_1 \cdot \\ \cdot \\ \cdot \\ \beta_m \end{align*} \text{endpar} \text{else} \text{par} \text{forall } 1 \leq i \leq n \\
\text{status}(\text{rcyes}_i) := \text{active} \\
\text{INSERT}(\alpha_i, \text{positions}(\text{rcyes}_i)) \text{forall } 1 \leq j \leq m \\
\text{status}(\text{rcno}_j) := \text{active} \\
\text{INSERT}(\beta_j, \text{positions}(\text{rcno}_j)) \]
Combination patterns with more than one, same-labeled outgoing edges including one to a mode are invalid (see Fig. 3.8 for one example pattern).

Figure 3.8: Example of a invalid pattern with a condition as source node. This is not valid because if a condition has an outgoing edge to a mode, this should be the only outgoing edge with this label.
4 Extension: Substitution

The previous specification covers only the basic Control State ASM constructs, constructs from other kind of ASMs like TurboASMs or constructs which are introduced with ASMs tools like CoreASM are not covered. Also the support to visualize the development process with ASM, like the ASM method, are not specified yet.

This section shows with an example how the specification can be extended by such constructs. In general, the definition looks like the basic specification definition. First the abstract syntax is given, often based on the one of the CSDs, then a concrete syntax is suggested and the last part is the definition of the semantics. As the first and second part look most time the same, there are in principle two possibilities for extending the specification in terms of behavior, as it is known in the case of ASMs: the extension of the translation rules or a preprocessor which first transforms the new constructs into combinations of known constructs.

The chosen example to show such an extension is about the possibility to visualize substitutions. Substitutions are used to make a CSD clearer, by outsourcing parts of it and rendering them more abstract in the original. A more precise definition, including functionality is given in Section 4.1. This extension uses a preprocessor for the semantics definition, which transforms the substitution diagrams to one big CSD, which can then be translated by the TRANSLATECSD ASM form the previous section.

4.1 Introduction

Substitutions based on the idea of refinements from the ASM method. In the ASM method the refinement principle is an important part to develop systems and software. Refinements are used in an iterative process to take an abstract part of an ASM and specify it to a more detailed part. In the ASM method this refinements must apply some conditions, for example if a state $S$ is manipulated by rules $\tau_1 \ldots \tau_n$ to a state $S'$ the whole sequence can be refined by an other sequence of rules $\sigma_1 \ldots \sigma_m$ which update a state $S^{*}$ to $S'^{*}$, limited by the condition that $S \equiv S^{*}$.
and $S' \equiv S^*$ where $\equiv$ means that the data in locations of interest in corresponding states are the same (see Figure 4.1) [9].

Based on these refinements, substitutions were developed to give a tool in CSDs to visualize abstract parts and further to simplify generally complex CSDs by abstraction possibilities. The following example is intended to clarify the meaning and purpose of substitutions.

In Figure 4.2 shows a CSD with about ten nodes inside. Such a CSD is clear and easy to understand. A CSD with about 100 nodes is very large and maybe not clear. If the complexity of such a large CSD is also high (when the nodes have multiple incoming and outgoing edges), the CSD is difficult to understand. In such cases, substitutions can help to reduce complexity by outsourcing a logically connected area into a separate diagram and replacing it with a single node or at least a smaller number of nodes in the original diagram. These abstracted areas should then specially be marked to show that this is the scope of the substitution.
4.1. INTRODUCTION

Another purpose of substitutions is that abstract diagrams which do not apply the conditions of a valid CSD can be created. Such abstract CSDs have the disadvantage that they are not translatable. More information about abstract CSDs are described later.

Derived from these purposes of substitutions, the extension needs a shape to mark an area in a CSD \((\text{SubstitutionArea} \ (\text{SA}))\) which is specified in detail in a separate diagram and, as special kind of such an area, a new kind of node \((\text{SubstitutionNode} \ (\text{SN}))\), which is an area with just one node and can be treated as a (abstract) rule where the rule node constraints need not be met.

For example when a rule should point to two modes A, B with the meaning that either the path to mode A is used or the other to mode B but the condition which is used to decide which path to go is inside a complex function. Normally a call to a complex function would be represented by a call rule node but by the conditions of rule nodes this is not allowed with two outgoing edges to modes. In such a scenario, a substitution can be used, like it is shown in Figure 4.3. As mentioned above, the conditions for rule nodes do not apply for substitutions (or more precisely for \(\text{SN}\)).

![Figure 4.3](image)

Figure 4.3: Example of a \(\text{SN}\) which represents an abstract node where the rule node conditions do not apply (Multiple outgoing edges to modes)

In cases where a \(\text{SN}\) applies to the conditions of rule nodes, the \(\text{SN}\) can also be replaced by an \(\text{SA}\), the result and the meaning would be the same. Figure 4.4 shows an abstract example where the \(\text{SN}\) has only one incoming and one outgoing edge (1), which mean that it fulfill the constraints of rule nodes, so it can be replaced by a rule node, and if needed, surrounded by an \(\text{SA}\) (2).

![Figure 4.4](image)

Figure 4.4: Example how an \(\text{SN}\) that applies to the rule node conditions (1) can be replaced by an \(\text{SA}\) (2) and vice versa

As already mentioned above, substitutions have their own conditions which must hold in a valid CSD with substitutions. Before these conditions are defined in more detail, a couple of
terms are introduced, which are often used in this section. For better understanding they are also illustrated in Figure 4.5.

![Diagram](image)

Figure 4.5: Example of a main diagram (1) with a marked sub-graph (2) (above) and a (refined) sub-diagram (3) (below)

**Main diagram (1):** Diagram where the substitution is marked. Mainly a CSD is the main diagram of a substitution. The main diagram contains the abstract part of a substitution.

**Sub-graph (2):** Part of a graph inside a main diagram. The sub-graph represents the abstract part of a substitution.

**Sub-diagram (3):** Separated diagram where the more detailed part of a substitution is shown. A sub-diagram can also be a main diagram for another substitution. A sub-diagram has markers which represents the incoming and outgoing edges (4).

With these terms the substitution constraints are defined by:

1. The sub-graph of a substitution must be one coherent graph with all intermediate nodes are part of the substitution
2. All nodes of a sub-diagram have to be valid
3. The number of incoming edges of the sub-diagram must be the same as the number of incoming edges of the sub-graph. Analog the number of outgoing edges of the sub-diagram must be the same as the number of outgoing edges of the sub-graph. ¹
4. If there are more then one incoming or outgoing edges, it had to be made clear, how the sub-diagram can be inserted into the main diagram. So the source of the incoming edges of a

¹The incoming and outgoing edges of a sub-graph are the edges which are connected to nodes outside of the sub-graph
sub-diagram should be mappable to the sources of the incoming nodes of the corresponding
sub-graph. The same applies to the outgoing edges and their targets.

5. All sub-graphs in a main diagram have to be disjoint

4.2 Substitution specification

The formal specification of substitutions is based on the CSD specification and extends it by a
couple of structural aspects, the corresponding representations, some constraints and the definition
of the semantics. The semantics definition is given by an Substitution2CSD ASM which works
as a preprocessor and transforms, if possible, all substitutions to one valid CSD.

The extension of the abstract syntax contains some modifications at the meta model. This
is extended by nodes for the substitution areas and nodes and a new class marker to mark the
incoming and outgoing edges of a substitution. Also a new subclass of a CSD for the sub-diagram
is added. The resulting metamodel is showed in Figure 4.6. The new parts are marked green.

For the formal definition, first the ASM of the basic CSD specification has to be extended.
Therefore some domains and functions are introduced.

- **domain** SubstitutionNode
- **domain** SubstitutionArea
- **domain** SubstitutionCSD
- **domain** Marker

Such a SubstitutionNode and SubstitutionArea contains a label which is a reference to a
SubstitutionCSD. The corresponding function to get this content is defined as follows:

\[ SNRef : SubstitutionNode \rightarrow String \]  
Function to get the reference to a sub-diagram as a String

\[ SARRef : SubstitutionArea \rightarrow String \]  
Function to get the reference to a sub-diagram as a String

SubstitutionAreas and SubstitutionNodes are grouped to Substitutions because both kind of
substitutions are often handled the same way.

- **domain** Substitution = SubstitutionArea ∪ SubstitutionNode
CHAPTER 4. EXTENSION: SUBSTITUTION

Figure 4.6: Metamodel of CSDs extended by substitutions (new parts are marked green)
As described above the SubstitutionCSD class is a subclass of CSD. The difference of a SubstitutionCSD and a CSD is that a SubstitutionCSD can have incoming and outgoing marker. To get the reference of a SubstitutionCSD, the following function can be used

\[
SCSDRef : \text{SubstitutionCSD} \rightarrow \text{String}
\]

Function to get the reference to a sub-graph as a String

Because it is often needed, the following function returns the SubstitutionCSD of a given reference or \textit{undef} if it not exists

\[
SCSDByRef : \text{String} \rightarrow \text{SubstitutionCSD} \cup \text{undef}
\]

Function to get a sub-graph by a reference

Marker are special edges with an additional attribute Type which describes if this is an \textit{InMarker} or an \textit{OutMarker}. They are used to mark the beginning and the end of SubstitutionCSDs and SubstitutionAreas. A Marker can be seen as a tuple (Edge, Type, Reference). If there are multiple \textit{InMarker} respectively \textit{OutMarker} in a SubstitutionCSD, the Marker are also connected with an source respectively target node outside of the SubstitutionCSD frame. This is necessary for the unique mapping of SubstitutionCSDs and the corresponding SubstitutionArea respectively SubstitutionNode.

The validation of substitutions is similar to the CSD validation. The functions contains the constraints from above and check if all Substitutions and SubstitutionCSDs are valid and ready to be transformed to a valid CSD.

\[
\text{ValidSubstitutions iff}
\]

\[
\text{ValidSubstitutionNodes and}
\]

\[
\text{ValidSubstitutionAreas and}
\]

\[
\text{ValidSubstitutionCSDs and}
\]

\[
\text{NoOverlappingSubstitutions}
\]

For substitution nodes a corresponding SubstitutionCSD had to exist and the number of incoming and outgoing edges had to be the same as in the corresponding SubstitutionCSD (see constraint 3). If there are more then one incoming or outgoing edges, it had also to be check if the back substitution is mappable (see constraint 4)
**CHAPTER 4. EXTENSION: SUBSTITUTION**

ValidSubstitutionNodes iff

forall \( sn \in \text{SubstitutionNode} \)

let \( scsd = \text{SCSDByRef}(\text{SNRef}(sn)) \) in

\( scsd \neq \text{undef} \) and \( |\text{Outgoing}(sn)| = |\text{OutMarker}(scsd)| \) and \( |\text{Incoming}(sn)| = |\text{InMarker}(scsd)| \)

\( \text{InMarker}(s) \) and \( \text{OutMarker}(s) \) are functions to get the Marker of SubstitutionCSDs and SubstitutionAreas.

\( \text{OutMarker}(s) = \)

if \( s \in \text{SubstitutionCSD} \) then

\( \{(e, \text{OUT}, \text{SCSDRef}(s)) | (e, \text{OUT}, \text{SCSDRef}(s)) \in \text{Marker}\} \)

else

\( \{(e, \text{OUT}, \text{SARef}(s)) | (e, \text{OUT}, \text{SARef}(s)) \in \text{Marker}\} - \text{InMarker}(s) = \)

if \( s \in \text{SubstitutionCSD} \) then

\( \{(e, \text{IN}, \text{SCSDRef}(s)) | (e, \text{IN}, \text{SCSDRef}(s)) \in \text{Marker}\} \)

x else

\( \{(e, \text{IN}, \text{SARef}(s)) | (e, \text{IN}, \text{SARef}(s)) \in \text{Marker}\} \)

\( \text{MappableSN}(sn) \) is a function to check if a SubstitutionCSD can be inserted at the position of the corresponding SubstitutionNode in the main diagram.

\( \text{MappableSN}(sn) \) iff

if \( |\text{Incoming}(sn)| > 1 \) then

forall \((s, t) \in \text{Incoming}(sn)\)

forsome \((s', t'), \text{IN}, \text{SCSDByRef}(\text{SNRef}(sn)) \) \( \in \text{InMarker}(\text{SCSDByRef}(\text{SNRef}(sn))) \)

\( s \equiv s' \)

if \( |\text{Outgoing}(sn)| > 1 \) then
forall \((s, t) \in \text{Outgoing}(sn)\)

forsome \(((s', t'), \text{OUT}, \text{SCSDByRef}(\text{SNRef}(sn))) \in \text{InMarker}(\text{SCSDByRef}(\text{SNRef}(sn)))\)

\(t \equiv t'\)

An equivalent function \(\text{MappableSA}(s)\) to check if a \(\text{SubstitutionCSD}\) can be inserted at the position of the corresponding \(\text{SubstitutionArea}\) in the main diagram.

\(\text{MappableSN(sn)}\) iff

if \(|\text{Incoming}(sn)| > 1\) then

forall \((s, t) \in \text{Incoming}(sn)\)

forsome \(((s', t'), \text{IN}, \text{SCSDByRef}(\text{SNRef}(sn))) \in \text{InMarker}(\text{SCSDByRef}(\text{SNRef}(sn)))\)

\(s \equiv s'\)

if \(|\text{Outgoing}(sn)| > 1\) then

forall \((s, t) \in \text{Outgoing}(sn)\)

forsome \(((s', t'), \text{OUT}, \text{SCSDByRef}(\text{SNRef}(sn))) \in \text{InMarker}(\text{SCSDByRef}(\text{SNRef}(sn)))\)

\(t \equiv t'\)

\(\text{SubstitutionAreas}\) have the same conditions as \(\text{SubstitutionNodes}\) and additional the sub-graph inside the area had to be coherent with all intermediate nodes has also to be inside the area (see constraint 1).

\(\text{ValidSubstitutionAreas}\) iff

forall \(sa \in \text{SubstitutionArea}\)

let \(scsd = \text{SCSDByRef}(\text{SNRef}(sa))\) in

\(\text{scsd} \neq \text{undef} \text{ and } |\text{OutMarker}(sa)| = |\text{OutMarker}(scsd)|\) and \(\text{InMarker}(sa)| = |\text{InMarker}(scsd)|\) and \(\text{Coherent}(sa)\) and \(\text{Mappable}(sa)\)
Chapter 4. Extension: Substitution

Coherent\((sa)\) checks if condition 1 holds. Therefore a depth-first search algorithm is used to look which nodes are reachable and if this are all the nodes in the area with no others in between.

\[
\text{Coherent}(sa) \text{ iff } \\
\text{forsome } n \in \text{nodes}(sa) \\
\text{ClearVisited()} \\
\text{Check}(n, sa) \\
\text{Visited} = \text{nodes}(sa) \\
\text{where } \text{Check}(n, sa) = \\
\text{if } n \in \text{nodes}(sa) \text{ then } \\
\text{InsertToVisited}(n) \\
\text{forall } s \in \text{Successor}(n) \text{ do } \\
\text{Check}(s, sa)
\]

SubstitutionCSDs have to apply the condition 2 which means that all nodes inside the sub-diagram have to be valid by the corresponding node conditions.

\[
\text{ValidSubstitutionCSDs iff } \\
\text{ValidModesAndRules and ValidConditions and ValidSuccessors and ValidEmbeddedCSDs and NoInvalidLoops and NoEmptyNodes}
\]

The functions ValidModesAndRules, ValidConditions, ValidSuccessors, ValidEmbeddedCSDs, NoInvalidLoops and NoEmptyNodes are the same functions as in Section 2.3.

The last validation checks if the substitutions are not overlapping. Therefore it looks if a node is at most inside one substitution.

\[
\text{NoOverlappingSubstitutions iff } \\
\text{forall } s_1, s_2 \in \text{Substitution} \\
\text{forall } n \in \text{nodes}(s_1) \\
\text{ } s_1 = s_2 \text{ or } n \notin \text{nodes}(s_2)
\]
The concrete syntax of SubstitutionNode and SubstitutionArea are defined by:

<table>
<thead>
<tr>
<th>Representation</th>
<th>Class name</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="DoSomething" /></td>
<td>Substitution node</td>
<td>A substitution rule is represented as dashed rectangle with an inner label. To get the content of the label, the function <code>substitutionNodeLabel</code> can be used.</td>
</tr>
<tr>
<td><img src="image" alt="DoRef" /></td>
<td>Substitution area</td>
<td>A substitution area is represented as dashed rectangle around a sub-graph of a diagram with an inner label. To get the content of the label, the function <code>substitutionAreaLabel</code> can be used.</td>
</tr>
</tbody>
</table>

A SubstitutionCSD is represented by a dashed frame with a header row which contains the name of the substitution\(^2\). An example of a SubstitutionCSD is showed in Figure 4.7.

![Figure 4.7: Example of a SubstitutionCSD. Unlike a CSD, the frame is dashed.](image)

The semantic of substitutions is defined by a preprocessor ASM Substitutions2CSD where all SubstitutionCSDs are inserted into the corresponding main diagram. Therefore the targets of the InMarker are connected to the sources of the corresponding edges in the main diagram, and vice versa for the OutMarker.

\[
\text{Substitutions2CSD} = \forall s \in \text{Substitution} \; \text{do} \\
\quad \text{let} \; \text{scsd} = \text{SubstitutionCSD}(s) \; \text{in} \\
\quad \text{if} \; s \in \text{SubstitutionArea} \; \text{then} \\
\quad \quad \text{MapSA(InMarker}(s), \text{InMarker}(\text{scsd}))
\]

\(^2\)which is equivalent to the reference
CHAPTER 4. EXTENSION: SUBSTITUTION

\[ MapSA(\text{OutMarker}(s), \text{OutMarker}(scsd)) \]
\[ \text{Delete(nodes}(s)) \]
\[ \text{Delete(edges}(s)) \]

else
\[ MapSN(\text{Incoming}(s), \text{InMarker}(scsd)) \]
\[ MapSN(\text{Outgoing}(s), \text{OutMarker}(scsd)) \]
\[ \text{Delete}(s) \]

where

\[ \text{SubstitutionCSD}(s) = \]
\[ \text{if } s \text{ in } \text{SubstitutionArea} \text{ then} \]
\[ \text{SCSDByRef}(\text{SARref}(s)) \]
\[ \text{else} \]
\[ \text{SCSDByRef}(\text{SNRef}(s)) \]
\[ \text{MapSN(edges, marker) =} \]
\[ \text{forall } ((s, t), \text{type}, \text{ref}) \in \text{marker do} \]
\[ \text{if type = IN then} \]
\[ \text{choose } ((s', t'), \text{ref}) \in \text{edges with } t' \equiv t \text{ do} \]
\[ \text{InsertEdge}(s', t) \]
\[ \text{DeleteEdge}(s', t') \]
\[ \text{else} \]
\[ \text{choose } ((s', t'), \text{ref}) \in \text{edges with } s' \equiv s \text{ do} \]
\[ \text{InsertEdge}(s, t') \]
\[ \text{DeleteEdge}(s', t') \]
\[ \text{MapSA(marker1, marker2) =} \]
\[ \text{forall } ((s, t), \text{type}, \text{ref}) \in \text{marker2 do} \]
\[ \text{if type = IN then} \]
\[ \text{choose } ((s', t'), \text{ref}) \in \text{marker1 with } t' \equiv t \text{ do} \]
\[ \text{InsertEdge}(s', t) \]
\[ \text{DeleteEdge}(s', t') \]
\[ \text{else} \]
\[ \text{choose } ((s', t'), \text{ref}) \in \text{marker1 with } s' \equiv s \text{ do} \]
\[ \text{InsertEdge}(s, t') \]
\[ \text{DeleteEdge}(s', t') \]
5 Evaluation

This chapter deals with the evaluation of the CSD specification. For this purpose, on the one hand a proof is given, which shows the completeness as well as the consistency of the specification based on the Basic ASMs. On the other hand, the implementation of a prototype will be described with the help of which CSDs can be generated and transformed into CoreASM code.

5.1 Proof of completeness and consistency

The first part of the evaluation of the specification is a proof to show that it is complete and consistent. First of all these terms are defined to get a common knowledge what they meant.

**Complete** means that each CSD with any combination of nodes can either be translated, or it can be shown at which point and why a translation is not possible.

**Consistent** means that the same graph or sub-graph has always the same translation, regardless of the context in which it occurs. Relative to the translator ASM, this means that no inconsistent update occur.

This proofs are only applied to the base specification of CSDs from Section 2 and 3. However, these can also be carried out analog for any extension.

First, the completeness property is shown by the sets of combination patterns for which a translation exists (see Section 3.2) and the set of combination patterns which are excluded by the constraints of the nodes and that the union of these sets is equivalent to the set of all combinations.

For the following description of sets, a special notation for the combination patterns is used. So in the following, a combination pattern is represented by a tuple \((x, y)\) which would mean that a single node \(x\) is connected with a single node \(y\). Because a node can have multiple incoming and outgoing edges, it should also be able to see \(x\) and \(y\) as sets of nodes. To differ single nodes from set of nodes, set of nodes are written with uppercase letters and single nodes with lowercase
CHAPTER 5. EVALUATION

letters. So for example \((x, Y)\) is the notation of a combination pattern with a single node \((x)\) as source node and multiple targets nodes \((Y)\).

To show the completeness property, first the set of all combination patterns is defined. This set contains all combinations where one node is directly connected to one or many other nodes. The single node can be both, the source or the target of the connections. Formally this set is described by Equation 5.1.

\[
\mathcal{C} = \{(n, M) \cup (M, n) \mid n \in \text{Node} \wedge M \subset \text{Node}\} \tag{5.1}
\]

Because a combination pattern with \(n\) source nodes can be split into \(n\) combination patterns with each has just a single source node (with each combination pattern has the same target node), these combination patterns are redundant and can be ignored because they are handled by the single source node combination patterns (see Figure 3.1 from Section 3.2).

As there are three kind of nodes (mode, rule and condition) (see Equation 5.2), the set \(\mathcal{C}\) can be split into three disjoint subsets \(\mathcal{M}, \mathcal{R}\) and \(\mathcal{C}\) (Equation 5.3), where each subset contains the combination patterns with the same source node. These subsets are defined by Equation 5.4, 5.5 and 5.6

\[
\text{Node} = \{\text{Mode}, \text{Rule}, \text{Condition}\} \tag{5.2}
\]
\[
\mathcal{C} = \mathcal{M} \cup \mathcal{R} \cup \mathcal{C} \quad \text{with} \quad \mathcal{M} \cap \mathcal{R} \cap \mathcal{C} = \emptyset \tag{5.3}
\]
\[
\mathcal{M} = \{(n, M) \mid n \in \text{Mode} \wedge M \subset \text{Node}\} \tag{5.4}
\]
\[
\mathcal{R} = \{(n, M) \mid n \in \text{Rule} \wedge M \subset \text{Node}\} \tag{5.5}
\]
\[
\mathcal{C} = \{(n, M) \mid n \in \text{Condition} \wedge M \subset \text{Node}\} \tag{5.6}
\]

The next step is, to divide these subsets also into disjoint subsets where either for all elements in a subset a translation exists or for all elements at least one condition can be named which is not complied.

The set of combination patterns with a mode as source node \(\mathcal{M}\) can be divided into three subsets \(\mathcal{M}_{RC}, \mathcal{M}_M\) and \(\mathcal{M}_M^+\) (Equation 5.7). \(\mathcal{M}_{RC}\) is the set of combination patterns where the targets nodes are either rules or conditions (or both of them) or the combination pattern is just a single mode node and is formally defined by Equation 5.8. \(\mathcal{M}_M\) contains the combination pattern with just one mode as target node (Equation 5.9). All elements of these subsets matches with
one of the combination pattern from Section 3.3, which mean that they are translatable. The last subset $M_{M^+}$ is one which contains combination patterns with multiple target nodes where at least one is a mode (Equation 5.10).

\[
M = M_{RC} \cup M_M \cup M_{M^+} \quad \text{with} \quad M_{RC} \cap M_M \cap M_{M^+} = \emptyset
\]  
\[(5.7)\]

\[
M_{RC} = \{(n, M) \mid n \in \text{Mode}, M \subset \text{Rule} \cup \text{Condition}\}
\]  
\[(5.8)\]

\[
M_M = \{(n, m) \mid n, m \in \text{Mode}\}
\]  
\[(5.9)\]

\[
M_{M^+} = \{(n, M) \mid n \in \text{Mode}, M \subset \text{Node} \wedge |M| \geq 2 \wedge \exists m \in M : m \in \text{Mode}\}
\]  
\[(5.10)\]

The last subset $M_{M^+}$ is by definition not translatable, because the combinations of this subset do not comply the constraint that a mode can just have an outgoing edge to a mode when this is the only outgoing edge.

The set of combination patterns with a rule as source node $R$ can be treated like $M$. It is divided into three subsets $R_{RC}, R_M$ and $R_{M^+}$ with the same definitions of the target nodes (Equation 5.11 - 5.14).

\[
R = R_{RC} \cup R_M \cup R_{M^+} \quad \text{with} \quad R_{RC} \cap R_M \cap R_{M^+} = \emptyset
\]  
\[(5.11)\]

\[
R_{RC} = \{(n, M) \mid n \in \text{Rule}, M \subset \text{Rule} \cup \text{Condition}\}
\]  
\[(5.12)\]

\[
R_M = \{(n, m) \mid n \in \text{Rule}, m \in \text{Mode}\}
\]  
\[(5.13)\]

\[
R_{M^+} = \{(n, M) \mid n \in \text{Rule}, M \subset \text{Node} \wedge |M| \geq 2 \wedge \exists m \in M : m \in \text{Mode}\}
\]  
\[(5.14)\]

Similar to $M_{RC}$ and $M_M$ the sets $R_{RC}$ and $R_M$ contain translatable combination patterns, because they match one combination pattern from Section 3.4. $R_{M^+}$ is not translatable because their combination patterns do not comply the constraint of outgoing edges to modes.

The set of combination patterns with a condition as source node $C$ is first divided into a subset that describes the combination pattern with a single condition $C_0$, which is defined by Equation 5.16 and a subset $C_+$ that contains all combination patterns with at least one target (Equation 5.17). These two subsets are by definition disjoint (see Equation 5.15).
\[ C = C_0 \cup C_+ \quad \text{with} \quad C_0 \cap C_+ = \emptyset \quad (5.15) \]

\[ C_0 = \{(n,) \mid n \in \text{Condition}\} \quad (5.16) \]

\[ C_+ = \{(n, M) \mid n \in \text{Condition}, M \subseteq \text{Node} \land |M| \geq 1\} = C_{+ \text{RC}} \cup C_{+ M} \cup C_{+ M^+} \quad (5.17) \]

Where

\[ C_{+ \text{RC}} \cap C_{+ M} \cap C_{+ M^+} = \emptyset \quad (5.18) \]

\[ C_{+ \text{RC}} = \{(n, M) \mid n \in \text{Condition}, M \subseteq \text{Rule} \cup \text{Condition} \land |M| \geq 1\} \quad (5.19) \]

\[ C_{+ M} = \{(n, M)_{\text{yes, no}} \mid n \in \text{Condition} \land \]

\( ((|M_{\text{yes}}| = 1 \land M_{\text{yes}} \subseteq \text{Mode} \land M_{\text{no}} \subseteq \text{Rule} \cup \text{Condition}) \lor \]

\( (|M_{\text{no}}| = 1 \land M_{\text{no}} \subseteq \text{Mode} \land M_{\text{yes}} \subseteq \text{Rule} \cup \text{Condition}) \lor \]

\( (|M_{\text{yes}}| = 1 \land M_{\text{yes}} \subseteq \text{Mode})) \} \quad (5.20) \]

\[ C_{+ M^+} = \{(n, M) \mid n \in \text{Condition}, M \subseteq \text{Node} \land \]

\( ((|M_{\text{yes}}| \geq 2 \land \exists m \in M_{\text{yes}} : m \in \text{Mode}) \lor \]

\( (|M_{\text{no}}| \geq 2 \land \exists m \in M_{\text{no}} : m \in \text{Mode})) \} \quad (5.21) \]

\[ M_{\text{yes}} \] are nodes, where the incoming edge from a condition is labeled with \textit{yes} respectively without a label, as no labeled outgoing edges from a condition have the same meaning as \textit{yes}-labeled edges. Analog, \( M_{\text{no}} \) are nodes, where the incoming edge from a condition is labeled with \textit{no}.

The combination pattern from \( C_0 \) is not translatable, because a condition have to have at least one outgoing edge by definition. Also the combination patterns from \( C_{+ M^+} \) are not translatable because, like the elements from \( R_{M^+} \) and \( M_{M^+} \), a condition can just have an outgoing edge to a mode when this is the only outgoing edge with this label.

The elements of the other subsets, \( C_{+ M} \) and \( C_{+ \text{RC}} \) matches one of the patterns from Section 3.5 and are therefore translatable.

Because the union of all disjoint subsets results in the set of all combination patterns \( C \) (see Equation 5.23) and for all subsets a translation or a reason why the elements are not translatable is given, the proof of completeness is fully proven.
\[ \mathcal{C} = M_{RC} \cup M_{M} \cup R_{RC} \cup R_{M} \cup C_{+RC} \cup C_{+M} \quad \text{translatable combinations} \]
\[ \cup M_{M^+} \cup R_{M^+} \cup C_{0} \cup C_{+M^+} \quad \text{not translatable combinations} \quad (5.23) \]

The second property, the consistency, is shown by an analysis of the combination patterns from the translations. First the consistency property is formally defined by:

\[ \forall p \in TP \, \exists! t \in T \quad (5.24) \]

where \( TP \) is the set of all translatable combination patterns and \( T \) is the set of function values of the translation function.

This statement is true when all combination patterns are pairwise different, formally

\[ \forall p_i, p_j \in TP \mid p_i \neq p_j, i \neq j \quad (5.25) \]

The set \( TP \) with all possible translatable combination pattern is shown in Figure 5.1.

So Equation 5.25 is applied on the whole set \( TP \) with a Brute-Force-Algorithm and it can be seen that it holds. And because Equation 5.25 holds, Equation 5.24 is also true, which means that the property of the consistency is fully proven.

5.2 Implementation

The second part of the evaluation is an implementation of a prototype with which CSD can be generated and transformed to CoreASM code. The implementation is inspired by the work of Piper Jackson [21] who implement a simple editor with the GMF based on a EMF [22] Ecore metamodel including an Eclipse Plugin to translate the generated diagram to CoreASM code. He used a simple model of CSDs with just modes, (abstract) rules and conditions.

The prototype, which was implemented within this work, also based on a metamodel which is defined by an EMF Ecore model. This model has a big difference compared to the model from the specification (Section 2): it defines edges between nodes as relation, this means that it do not contain a separate edge class, but \( \text{succ} \) and \( \text{prev} \) relations between the nodes which represent the successor (\( \text{succ} \)) and previous (\( \text{prev} \)) nodes. Together with this difference, the condition node class
Figure 5.1: Possible combinations of nodes in a CSD
5.2. IMPLEMENTATION

needs other relations to the other nodes to map the two possible paths (yes- and no-paths). These
relations are modeled by the yesSucc and noSucc relations. Figure 5.2 shows the class diagram
representation of this model with all differences to the specification metamodel are marked.

The graphical editor is specified with the Eclipse Sirius Project [23] which is an alternative to
GMF. Sirius based on EMF/GMF and is still being actively developed, in contrast to GMF which
last update was in 2011 [24], [25]. The Sirius part contains a specification of the diagram editor.
This includes the representations of the different kind of nodes and the definition of the toolbar.
Figure 5.3 shows a screenshot of this specification containing a structure which shows what is
defined. With Sirius, it is possible to define nodes which just contain text, like call or update rule
nodes for this work, and container, which can contain some nested nodes. Such container were
defined to represent rule nodes which can contain an EmbeddedCSD, like forall rule nodes.

With this definition of an Ecore model and a Sirius specification, it is possible to generate and
edit CSDs. In order to provide even more functionality, an eclipse plug-in has been developed
which takes a diagram and translates it into CoreASM code. EMF/Sirius provide a diagram in
a XML-like format which is first parsed into a CSD data object with a SAXParser [26]. This
parsed object is then taken by a translator which produces a CoreASM code skeleton. The
actual implementation of the translator is a one-to-one implementation of the semantics definition
of the CSD specification (Section 3.2). Because the TRANSLATECSD ASM from the specification
is often very abstract, e.g. with the definition of the functionality of the position or the definition
of the translation object, first an interface based skeleton was implemented. An example of these
interfaces is printed in Listing A.5, the other can be found in Section A. In a final step these
interfaces are used to implement the translator. The translation object was implemented as an
object tree where the child nodes are inner nodes1. The position is in this case a reference2 which
is set to a tree node.

```java
package de.maleitz.translator.Interfaces;

import de.maleitz.translator.model.*;

public interface ITranslator {
    String translateCSD(CSD csd);

    void translate(Node n, IPosition pos, ITranslateObject translation);
}
```

Listing 5.1: Interface of the translator

1 this means that in the resulting code, these parts are indented
2 as a Universally Unique Identifier (UUID)
Figure 5.2: Ecore metamodel. The successor and previous relations (red ellipses) are different to the specification metamodel.
Figure 5.3: Screenshot of the Sirius specification for the CSD generation and translation tool
CHAPTER 5. EVALUATION

The constraint aspects, which are handled in the specification by the validation predictions (Section 2.3) are handled in the implementation by Sirius validation rules and a validation method which is called on the parsed CSD object. A third validation method which extend the Ecore model by OCL constraints was not implemented, because the other two validations covers all constraints and gives the user the most freedom during the creation of a CSD, because the validation is triggered by the user (Sirius validation rules, see screenshot in Figure 5.4), respectively by the translator (CSD validation method).

![Figure 5.4: Screenshot of the Sirius validation triggered by the user via the context menu](image)

An alternative to the plug-in translator would have been an Acceleo implementation. Acceleo is a Model-to-Text language, which enables the user to generate code from a given model instance. The approach with the plug-in translator was chosen because it is better suited to evaluate the specification, which was the purpose of this implementation.
6 Discussion

During the development of the specification a couple of discussions and design decisions were made. This sections is about this discussions and the reasonings of the decisions made. This discussions are grouped by discussion about the basic specifications, discussions about the substitution extension and the discussion about how the specification can be evaluated.

6.1 Basic Specification

The work on this thesis began with an evaluation of the current situation of the graphical representation of Control State ASMs. As already mentioned in the motivation (see Section 1.1) the main problem that emerged from this evaluation is that there is no complete specification. So the main work was the re-engineering of the partial specifications like in [2] and to develop a full formal specification. Finally, a model-based approach was chosen for the work, which on the one hand has the advantage that there is a clear overview of the CSDs, which simplifies the extension of the specification. On the other hand, such an approach is best suited for the later implementation of the CSD generating and translating tool.

Because there were a lot of discussion and decision points, they are grouped by the four known aspects from the definition of a specification. When a discussion covers more then one aspect topic, it is handled in the topic which matches it most.

6.1.1 Structural Aspects

Since it was clear that a model-based approach for the definition of the specification is used, a class-diagram-based meta model was initially made.

The basic structural aspects like abstract nodes and edges were derived from the fact that a CSD is a simple directed graph. The ASM theory together with the common representation of FSMs (different kind of flowcharts) and the existing diagrams in the current literature, inspired
the next level of the structure, modes, rules and conditions. The concrete rule nodes were taken over by the basic ASM definition [2].

A main question at this point was, whether conditions should be seen as a subclass of rules, or as their own kind of node. The ASM theory says that conditions are a kind of rule, so the first idea was that the rule branch contains conditions as a child node, as it is shown in Figure 6.1. On the other side, in all existing diagrams conditions have a different representation from rules.

![Diagram of alternative meta model with conditions as a child node of rules.](image)

Figure 6.1: Detail of the alternative meta model with conditions as a child node of rules

Additionally, unlike rules, conditions do not change the signature of an ASM so the meaning and translation of a condition is not similar to other rules. Because of the last point the final structure for conditions was chosen. This has also the positive side effect that the constraint handling (with conditions) and the proof of completeness and consistency is easier.

Another discussion point on the structural aspects was the question how the start of a diagram is marked. This is necessary because often Control State ASMs are cyclic without an explicit start, like the simple traffic light example (see Section 1.1), the corresponding CSD without a marker is shown in Figure 6.2. Here it is hard to say, where a translation algorithm should start.

![Diagram of CSD for simple traffic light example.](image)

Figure 6.2: CSD of the simple traffic light example from Section 1.1
During the development a couple of ideas were tested. First, the start, which by definition had to be a mode, is an attribute, which is set at exactly one mode per CSD. This special mode is then later represented in a special way. The alternative was a new kind of node, called startpoint, which has exactly one outgoing edge to the start mode and no incoming edge. With respect to the simplicity the attribute solution was chosen. The representation aspect of this attribute is discussed later on in Section 6.1.3.

The last discussion on the structural aspects was about the body of let-, forall-, choose- and parallel-rules. In the ASM theory these rules describe in their body something like a scope, which consists of at least one rule. The simplest idea, based on the introduction of the forall and choose rule representation in [2], was to just allow exactly one rule in the body of these kind of rules. If the user wanted more than one rule in a body, he had to use a call rule, and specify the body in a separate rule. Another idea was to define a subclass of CSDs, called EmbeddedCSD, which represents the subgraph inside of such a body. The advantage of such an EmbeddedCSD is that both, a single rule and a complex subgraph can be represented inside a body. The third alternative was to see let-, forall-, choose- and parallel-rules as a scope with a special header, which means that a kind of frame surrounds the nodes which should be inside the body. Figure 6.3 shows an example how this could look like. This would have the same advantages like the second idea, but an estimation of the translation complexity shows that the second idea with the EmbeddedCSDs is simpler at the translation process with the same opportunities and the same level of understandability, usability and scalability.

6.1.2 Constraint Aspects

The discussion on the constraints aspects were mainly about the valid number of incoming and outgoing edges. This discussion is directly connected with the behavioral aspects of CSDs so both aspects are discussed here. The first thought was, based on the existing diagrams, that only
modes can have no outgoing edge. This was simply obvious because a Control State Rule always ends with the update of the mode. Later, with the introduction of EmbeddedCSDs, also rules can be the end of a sub-graph. So modes and rules can both have no outgoing edge, while conditions have at least one outgoing edge, because a condition without any actions for at least the positive or the negative evaluation is pretty useless. The upper bound of outgoing edges was first chosen as one. Because many diagrams use multiple outgoing edges to represent parallel execution, this should be also a possibility beside the parallel node. So, finally it was decided that in general no node has a limited number of outgoing edges. Only in case of modes in EmbeddedCSDs no outgoing edges are allowed. This has to do with the fact that within e.g. a forall rule, a new Control State rule would start and this does not conform to the definition of Control State ASMs.

Analog discussions were then also made on the number of incoming edges. First, before the start marker and EmbeddedCSD were introduced rules and conditions had to have at least one incoming edge, because, as mentioned above, a Control State Rule is always bounded by modes. For modes on the other side, it was not clear if they can have no incoming edges, until the start marking problem was solved. The main discussion on the number of edges was about the upper bound of incoming edges of nodes in general. The question was: Is it necessary to consider the problem in what context multiple incoming edges stand? Or in other words: Is it necessary to mark the end of e.g. a par split? For example if a rule A has multiple outgoing edges (par split) and they are later (finally) merged in rule F as it can be seen in Figure 6.4. In which context stands the next rule G? Is F the end of the par split and G is sequential to this par block or is there no end and G is executed three times, as the par split has three paths (A – B – E – F – G; A – C – E – F – G; A – D – F – G). With the first idea some more problems occur. For example what if not all paths of a par split are merged together, like in rule E? This can not be an end of the whole par split, but it is a merge like in rule F. So with this context, the par split should be read as two splits, \((B \text{ par } C) \text{ par } D\).

![Figure 6.4: Part of a schematic example CSD with one parallel splits from A to B, C, D and some partial merges B, C to E and D, E to F.](image)

Although this is already very complex, it should also noted that two paths do not necessarily have to be in a parallel relation, e.g. if they come from a condition and each represent the yes
6.1. BASIC SPECIFICATION

<table>
<thead>
<tr>
<th>Edge Type</th>
<th>Number of incoming edges</th>
<th>Number of outgoing edges</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mode</td>
<td>0..*</td>
<td>0..* / 0 if inside EmbeddedCSD</td>
</tr>
<tr>
<td>Rule</td>
<td>0..*</td>
<td>0..*</td>
</tr>
<tr>
<td>Condition</td>
<td>0..*</td>
<td>1..*</td>
</tr>
</tbody>
</table>

Table 6.1: Number of incoming and outgoing edges from the different node types

respectively no path. Then the merge is not an end par. So for the sake of simplicity the second idea was chosen, which may lead to irritation for users with prior knowledge of state charts and BPMN, but is better in terms of the complexity of algorithms and still understandable for all users. With this decision and the introduction of EmbeddedCSDs, there are no constraints about the number of incoming edges of any node needed.

Table 6.1 shows the final numbers of incoming and outgoing edges of the different node types.

Because the number of incoming and outgoing edges of rules to other rules and conditions is limited only by a lower bound (0), quite complex CSDs can be modeled. Figure 6.5 shows such a case with various splits and merges. Listing 6.1 contains the corresponding translation.

```plaintext
if mode = m1 then
  par
  seq
    rule1
  par
    seq
      rule2
      rule5
    par
      seq
        rule7
        rule10
      endseq
    seq
      rule8
      rule10
```

Figure 6.5: Part of a complex example CSD with some parallel splits and some various merges.
Another discussion about the constraints was the *NoInvalidLoops* validation. This is needed because a loop between two modes, like in Figure 6.6, is not translatable, because the two rules \textit{rule}_1 and \textit{rule}_2 theoretically have to be written infinitely often in succession and therefore the translation algorithm would not terminate. There has been much discussion on how this issue could possibly be solved more elegantly, but then decided that this is not part of this work or beyond the scope of this work. So a simple depth-first-search algorithm is used to check for such loops and if one loop is founded the CSD is invalid.

Listing 6.1: Translation of the complex CSD example in Figure 6.5
6.1.3 Representation Aspects

The discussion on the representational aspects of CSDs are, like the discussion on the constraints, directly connected with the discussions about their behavior. Remember that the representations in this thesis are just one kind of possible visualizations, which is derived from the abstract syntax and that the goal of this thesis is not to give the best concrete syntax, but define the formal specification of CSDs with just one possible concrete syntax. Therefore the discussions were mainly more abstract and describes e.g. the design relative to other kind of nodes and not on the concrete representation.

One of the earlier discussion occurs during the literature review about related work. None of the previous specifications or their approaches dealt with the presentation of all basic ASM rules. The work with the highest coverage was [2] where modes, conditions, (abstract) rules, forall- and choose rules were introduced. So how should the other rules be represented? First of all, call, update and skip rules can simply be represented as the (abstract) rules with the difference of the label. A let rule is, as mentioned above, similar to forall and choose, as it has a header and something like a scope, so the representation should also be similar where these two parts are separated. For the parallel block rule a design was chosen where the rules are painted one below the other, which is similar to the parallel execution by multiple outgoing edges where the successor nodes are mostly also painted one below the other.

The last rule of the basic ASM constructs which had no graphical representation was the sequential rule. During the work on this thesis, a lot of discussions were made about how such sequential blocks can be represented. The first idea was a shape similar to parallel rules with the difference that the rules should be split in a different way. Figure 6.7 shows a concept of such a shape. The interpretation of this shape was that the rule in the top row, \textit{rule}_1, is executed first and then the rule in the second row, \textit{rule}_2 and so on. During the discussions the argument
often came up that this is not a common representation for sequential execution, and this much more arises about the sequence of connected nodes. Therefore, the idea with the extra shape was discarded. The next question was, should there be another kind of edge to distinguish between parallel and sequential execution? The advantage would be, that with different edges, a parallel block have not to be represented by the parallel rule node or by multiple outgoing edges, but can also be represented by a sequence of nodes connected by parallel edges. On the other side, this could result in complex nested node sequences, which are confusing and hard to understand, like the example in Figure 6.8 shows. One kind of edge with different meanings depending on the number of edges with the same source node was decided to be the best solution. It is similar to other diagrams like flowcharts and also covers both, parallel and sequential execution in a clear and well-defined manner.

Figure 6.8: Example of a schematic CSD with two types of edges. The merge with the different edges types in mode $m_2$ make this example difficult to translate

The last point about the representation discussion was the visualization of the $startNode$ and $initialMode$ attribute. In other words, how can the entry point of a diagram be represented. Many alternatives have been proposed, and all would be acceptable, ultimately, these nodes should somehow be different from the others, but the node type should still be recognizable. Figure 6.9 shows some of the suggestions that were discussed.

Figure 6.9: Some discussed alternatives for the $startNode$ and $initialMode$ representation

6.1.4 Behavioral Aspects

The discussions about the behavioral aspects are mainly about how a CSD can be translated to ASM code and which properties should be complied, like no inconsistent updates and no needless nesting in the resulted ASM code.

The translation process should be so simple that a reader can systematically translate a CSD as quickly as possible and without any additional resources. Only when this is done can a CSD fulfill the purpose of making an ASM more understandable through a graphical representation.
Therefore, one requirement of the translation algorithm is that it be as systematic and simple as possible. This is best done by going through the CSD node-by-node and translating the actual node. This is therefore also the approach for the chosen algorithm. Since the translation of a node is dependent on the meaning of its outgoing edges, the successor nodes must always be considered to know if the edges require parallel or sequential execution after the actual node. So the algorithm should include translations for all valid combinations from a single node to any number of successor nodes. Since the number of combinations is endless because each node can have any number of outgoing edges, these have been grouped and a matching pattern developed for each group which describes the corresponding combinations.

Because a CSD can be a cyclic graph, the algorithm should also remember which nodes it has already considered. Therefore the attribute \textit{status} was added. Because a node can be reached by different paths (when it has multiple incoming edges) the algorithm should notice that the translation of this node had to be added to the translated text multiple times. This is handled by \textit{Position} markers in the translated text, which are inserted for every incoming edge. The exact definition of a position has been deliberately omitted because it is not relevant to the specification as it will be implemented later. This would also depend on how the translation text object is exactly implemented. Since this was also left abstract, since this is also not relevant for the specification, no exact definition of the position could be given. Possible solutions would be e.g. an abstract syntax tree (AST) for the translation text object, then the position would correspond to an "empty" node in that tree with a unique reference which is later replaced by a part of a translation.

It was decided that the resulted ASM code have not to comply constraints like that there are no inconsistent updates or that the code do not contain nested parallel or sequential blocks which could be written as one block. This decision was made because the goal of this thesis is not an optimal interpreter of CSDs, but a formal specification of them.

6.2 Substitution Extension

This section is about the substitution extension of the CSD specification in Section 4. The main purpose of this specification was to show how the basic specification of CSDs can be extended to other ASM classes like TurboASMs or MultiagentASMs.

In general there are, like in ASMs, two ways to extend the specification, by adding a kind of preprocessor which transforms the new structures to known structures or by extending the translation rules.
The idea behind the chosen extension was not to take another class of ASMs but to use the ASM method as the basis for an extension, with which the development process of an ASM can be documented.

With this idea, the first approach includes different classes of substitutions which comply different constraints of the ASM method and other extensions of ASM tools like the Extension Plugin in CoreASM [6]. Because this would result in a very complex extension with multiple shapes and constraints which would go beyond the scope of this section and this thesis in general, a more abstract approach was chosen. The goal of this approach was to give the opportunity to mark a section in a CSD which is specified in another diagram in more detail.

The next question about the substitutions was, which constraints a valid substitution and the corresponding SubstitutionCSD should fulfill? First of all, a substitution should always be clearly replaced by an SubstitutionCSD. This means that for every InMarker of a SubstitutionCSD exactly one InMarker of the corresponding Substitution have to exist. Analog, for every OutMarker of a SubstitutionCSD exactly one OutMarker of the corresponding Substitution have to exist. To check this constraint and later to replace the Substitution by the SubstitutionCSD, a simple depth-first-search algorithm was used. An optimization of this algorithm was also a discussion point, which was quickly rejected as this would in turn go beyond the scope of the thesis and the goal is indeed only a first, complete specification and not that this is also optimal everywhere.

Another constraint of substitutions was that they should not be overlapping. This constraint is needed because there is no prioritization at the preprocessing where the SubstitutionCSD are back substituted. This could result in different CSDs or even a mistake in the back-substitution process. Figure 6.10 shows schematically a starting situation for such a case. The substitutions a and b overlap in Rule 2, the corresponding SubstitutionCSDs are also shown schematically in Figure 6.10. Figure 6.11 now shows the back substitution where a is first replaced and then b. In this case, both substitutions are easily feasible. Figure 6.12 shows the reverse order of the back substitutions. The replacement of a is not clearly possible after that of b, because the number of outgoing edges does not match. In the example, the number of outgoing edges was implicitly increased. However, it will be appreciated that the different orders of back substitution of the overlapping substitutions give different results and should therefore be excluded for the CSD extension.
6.2. SUBSTITUTION EXTENSION

Figure 6.10: Schematic CSD with two overlapping substitutions $a$ and $b$ (top) and the corresponding SubstitutionCSDs (middle and bottom)

Figure 6.11: Back substitution of first $a$ and then $b$. In this case, there are no problems during the back substitutions
Figure 6.12: Back substitution of first $b$ and then $a$. After $b$ is replaced, the substitution of $a$ is not valid because of the number of OutMaker is different. For this example, the number of OutMarker in the SubstitutionCSD $a$ was implicitly increased, so that it matches

6.3 Evaluation

At the beginning of the work it was also discussed how the specification can be evaluated at the end. There were two different opinions. On the one hand, a theoretical evaluation was demanded, because the formal specification largely provides only a theoretical basis for the later use of CSDs. The other opinion called for a practical application to show that this specification is also applicable in practice. It was then decided to satisfy both parties by providing evidence demonstrating completeness and consistency and developing a prototype to generate CSD and transform it into CoreASM code.

For the proof, the definitions of completeness and consistency mentioned in Section 5.1 were first derived and then shown in a formal mathematical quantity proof.

For the prototype, an existing skeleton was taken from Jackson Piper and adapted to the specification, as described in Section 5.2.
7 Conclusion

In this thesis a formal specification of Control State Diagrams was defined. For this purpose, a model-based approach was used, which includes the definition of the abstract syntax, a derived concrete syntax and the semantics.

During the work, no care was taken to create an optimal specification, but rather to create a basis for generating, in terms of their semantics, uniform CSDs, which thereby can be easily understood by any reader with the background knowledge of this specification.

In order to prove its suitability for practice, the evaluation has in addition of a proof of completeness and consistency also developed a prototype of a tool with which CSDs can be generated and transformed into CoreASM code.

Finally, it has been shown by an example, how the specification can be extended to be e.g. later also visualize other classes of ASMs.

7.1 Future Work

In future work, on the one hand, the translation algorithm from section 3 could be optimized in terms of runtime. This applies above all to the validation functions such as NoInvalidLoops or NoOverlappingSubstitutions from the extension (see Sec.4).

Another optimization that has already been discussed in the context of this thesis is the check for inconsistent updates during the translation process or even during the creation of the diagram, i.e. a kind of check of CSDs for inconsistent updates.

One last optimization that is currently under discussion is the resolution of nested parallel blocks or sequential blocks. This includes the optimization of the pattern (.rule1. rule) with a rule as a source node and a rule or condition as the only successor. Actually this is translated as:
The better translation would be:

```plaintext
seq
  rule1
par
  rc
endpar
endseq
```

So the parallel block is not started because it contains just one element.

On the other hand, the specification could be extended by other ASM classes, so that the specification covers any kind of ASM. It is also possible to carry out studies investigating the usability of CSDs. Also, work on the compatibility of the ASM method with the substitution extension could be interesting for the future, which could result in how the ASM method can best be visualized with CSDs.

Another future work could be the improvement of the implementation. This contains the optimizations from above and also the implementation of the substitution extension. In general the GUI and the usability of the prototype could be improved, so that it becomes more user-friendly.
A Implementation

```java
package de.maleitz.translator.Interfaces;

import java.util.UUID;

public interface IPosition {
    UUID getRef();
}
Listing A.1: Interface of the position

package de.maleitz.translator.Interfaces;

public interface ITranslateObject {
    void add(ITranslatePart part, IPosition pos);
    ITranslatePart searchPart(IPosition pos);
    String toString();
}
Listing A.2: Interface of the translation object

package de.maleitz.translator.Interfaces;

public interface ITranslatePart {
    void change(ITranslatePart newContent);
    String getPreContent();
    String getPostContent();
}
Listing A.3: Interface of the element of a translation object
```

package de.maleitz.translator.Interfaces;

83
import de.maleitz.translator.model.Rule;

import java.util.List;

public interface ITranslationPattern {

  ITranslatePart modeSingle(String modeLabel);
  ITranslatePart modeOneMode(String sourceModeLabel, String targetModeLabel);
  ITranslatePart modeRCs(List<IPosition> rcPositions, String modeLabel);
  ITranslatePart translateRule(Rule rule);
  ITranslatePart ruleSingle(Rule r);
  ITranslatePart ruleOneMode(String modeLabel, Rule r);
  ITranslatePart ruleRCs(List<IPosition> rcPositions, Rule r);
  ITranslatePart conditionOneMode(String modeLabel, String conditionLabel);
  ITranslatePart conditionTwoMode(String yesModeLabel, String noModeLabel, String conditionLabel);
  ITranslatePart conditionRCsYes(List<IPosition> rcPositions, String conditionLabel);
  ITranslatePart conditionRCsYesNo(List<IPosition> yesRCPositions, List<IPosition> noRCPositions, String conditionLabel);
}
ITranslatePart conditionRCsYesModeNo(List<IPosition> rcPositions,
String modeLabel, String conditionLabel);

ITranslatePart conditionModeYesRCsNo(List<IPosition> rcPositions,
String modeLabel, String conditionLabel);
}

Listing A.4: Interface of the translation patterns

package de.maleitz.translator.Interfaces;

import de.maleitz.translator.model.*;

public interface ITranslator {
    String translateCSD(CSD csd);
    void translate(Node n, IPosition pos, ITranslateObject translation);
}

Listing A.5: Interface of the translator
Figure A.1: Screenshot of the diagram editor. In the menu bar is an extra button to start the translation process.
Figure A.2: Screenshot of the diagram editor after Sirius validation. An error is marked and listed in the problems view.
Figure A.3: Example of a validation rule in Sirius. It validates that if a mode is a successor node, no other successor nodes exists.
List of Figures

1.1 CSD of an refined ATM model [2, p. 98, Fig. 3.2] ......... 3
1.2 CSD of a daemon game ASM [2, p. 51, Fig. 2.13] ............. 3
1.3 Example CSD with an other syntax which is similar to the BPMN notation [3, Fig. 2] .................. 4
1.4 Introduction of control state diagrams in [2, p. 45, Fig. 2.5] ......... 5
1.5 Alternative introduction of control state diagrams in [2, p. 45, Fig. 2.6] ............. 6
1.6 Suggested CSD construct for a forall and a choose rule [2] ............. 6
1.7 Aspects of a computer language description [17, Fig. 1] ............. 11

2.1 Metamodel CSD ........................................ 25

3.1 Handling of patterns with multiple incoming edges .................. 28
3.2 Multiple outgoing edges pattern ............................ 28
3.3 Example of a translation rule with the graphical combination pattern of a rule to a mode node on the left side and the corresponding translation actions in the right side. .......................................................... 29
3.4 Example of a translation. Translated text is written in monospace font (3). Functions which must be first evaluated are written in italic font (1). α is the position marker where the next translation should be inserted (2). .......... 30
3.5 Multiple incoming edges example ............................... 32
3.6 Not valid patterns with a mode as source node. This pattern is not valid because if a mode has an outgoing edge to a mode, this should be the only outgoing edge. 36
3.7 Not valid patterns with a rule as source node. This pattern is not valid because if a rule has an outgoing edge to a mode, this should be the only outgoing edge. .... 40
3.8 Example of an invalid pattern with a condition as source node. This is not valid because if a condition has an outgoing edge to a mode, this should be the only outgoing edge with this label. .......................................................... 45

4.1 Visualization of legal refinement [2] ............................. 48
4.2 Simple CSD with ten nodes ................................. 48
List of Figures

4.3 Example of a SN which represents an abstract node where the rule node conditions
do not apply (Multiple outgoing edges to modes) ........................................... 49
4.4 Example how an SN that applies to the rule node conditions (1) can be replaced
by an SA (2) and vice versa ................................................................. 49
4.5 Example of a main diagram (1) with a marked sub-graph (2) (above) and a (refined)
sub-diagram (3) (below) ........................................................................ 50
4.6 Metamodel of CSDs extended by substitutions (new parts are marked green) .... 52
4.7 Example of a SubstitutionCSD. Unlike a CSD, the frame is dashed. ............... 57
5.1 Possible combinations of nodes in a CSD ................................................. 64
5.2 Ecore metamodel .................................................................................. 66
5.3 Sirius specification ................................................................................. 67
5.4 Sirius validation .................................................................................... 68
6.1 Section of an alternative metamodel ....................................................... 70
6.2 CSD of the simple traffic light example from Section 1.1 ......................... 70
6.3 Forall shape as scope ........................................................................... 71
6.4 CSD with split and partial merge ........................................................... 72
6.5 Part of a complex example CSD with some parallel splits and some various merges. 73
6.6 Invalid Loop ....................................................................................... 75
6.7 Sequential rule node concept .................................................................. 75
6.8 Example of a CSD with two type of edges ............................................. 76
6.9 startNode and initialMode representations ............................................. 76
6.10 Overlapping Substitution 1 .................................................................. 79
6.11 Overlapping Substitution 2 .................................................................. 79
6.12 Overlapping Substitution 3 .................................................................. 80
A.1 Screenshot of the diagram editor ............................................................ 86
A.2 Screenshot of the diagram editor with error ........................................... 87
List of Tables

1.1 Basic ASM rules [2] .................................................. 9

6.1 Number of incoming and outgoing edges from the different node types .......... 73
Bibliography


Erklärung
Ich erkläre, dass ich die Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Ulm, den ..................................................

Markus Leitz