SoCle project

UML and OCL Semantics in ASM

John Mullins       Mathieu Bergeron

December 14, 2004

ÉCOLE POLYTECHNIQUE MONTRÉAL
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>OCL Semantics</td>
<td>38</td>
</tr>
<tr>
<td>4.1</td>
<td>Expressions</td>
<td>38</td>
</tr>
<tr>
<td>4.1.1</td>
<td>Syntax</td>
<td>38</td>
</tr>
<tr>
<td>4.1.2</td>
<td>Semantics</td>
<td>39</td>
</tr>
<tr>
<td>4.2</td>
<td>Constraints</td>
<td>41</td>
</tr>
<tr>
<td>4.2.1</td>
<td>Syntax</td>
<td>41</td>
</tr>
<tr>
<td>4.2.2</td>
<td>Semantics</td>
<td>42</td>
</tr>
<tr>
<td>4.2.3</td>
<td>Templates</td>
<td>43</td>
</tr>
<tr>
<td>4.3</td>
<td>UML Model Semantics Extensions</td>
<td>44</td>
</tr>
<tr>
<td>A</td>
<td>ASM Details</td>
<td>47</td>
</tr>
<tr>
<td>A.1</td>
<td>Term Evaluation</td>
<td>47</td>
</tr>
<tr>
<td>A.2</td>
<td>Domains</td>
<td>48</td>
</tr>
<tr>
<td>A.3</td>
<td>Predicate Semantics</td>
<td>48</td>
</tr>
<tr>
<td>A.4</td>
<td>Rule Semantics</td>
<td>48</td>
</tr>
<tr>
<td>A.5</td>
<td>External Functions</td>
<td>49</td>
</tr>
<tr>
<td>A.6</td>
<td>Extend Rule</td>
<td>51</td>
</tr>
<tr>
<td>B</td>
<td>UML Semantics Details</td>
<td>52</td>
</tr>
<tr>
<td>B.1</td>
<td>Static Vocabulary</td>
<td>52</td>
</tr>
<tr>
<td>B.1.1</td>
<td>Extracted Information</td>
<td>52</td>
</tr>
<tr>
<td>B.1.2</td>
<td>Elaborated Information</td>
<td>54</td>
</tr>
<tr>
<td>B.1.3</td>
<td>Well-Formedness</td>
<td>58</td>
</tr>
<tr>
<td>B.1.4</td>
<td>Method Call Actions</td>
<td>62</td>
</tr>
<tr>
<td>B.2</td>
<td>Configuration Vocabulary</td>
<td>63</td>
</tr>
<tr>
<td>B.2.1</td>
<td>Extracted Information</td>
<td>63</td>
</tr>
<tr>
<td>B.2.2</td>
<td>Well-Formedness</td>
<td>63</td>
</tr>
<tr>
<td>B.2.3</td>
<td>Initial Configuration</td>
<td>64</td>
</tr>
<tr>
<td>B.3</td>
<td>Additional Vocabulary</td>
<td>65</td>
</tr>
<tr>
<td>B.3.1</td>
<td>Well-Formedness</td>
<td>66</td>
</tr>
<tr>
<td>B.4</td>
<td>Transition Rule</td>
<td>67</td>
</tr>
<tr>
<td>B.4.1</td>
<td>Choosing a Thread</td>
<td>67</td>
</tr>
<tr>
<td>B.4.2</td>
<td>Collecting Enabled Transitions</td>
<td>67</td>
</tr>
<tr>
<td>B.4.3</td>
<td>Selecting Transitions to Fire</td>
<td>70</td>
</tr>
<tr>
<td>B.4.4</td>
<td>Firing Transitions</td>
<td>72</td>
</tr>
<tr>
<td>B.4.5</td>
<td>Applying Updates</td>
<td>75</td>
</tr>
<tr>
<td>C</td>
<td>OCL Semantics Details</td>
<td>77</td>
</tr>
<tr>
<td>C.1</td>
<td>OCL Expressions Type System</td>
<td>77</td>
</tr>
<tr>
<td>C.2</td>
<td>μ-Calculus Semantics</td>
<td>77</td>
</tr>
</tbody>
</table>
List of Figures

1.1 UML Class Diagram .......................................................... 7
1.2 UML Object Diagram .......................................................... 7
1.3 UML Statechart Diagram ...................................................... 8
1.4 Example of an Invariant ....................................................... 9
1.5 Example of a Postcondition ................................................... 10
1.6 Example of a Memory Cell .................................................... 10
1.7 Cell’s Behavior ................................................................. 11
1.8 Extended Memory Cell ......................................................... 12
1.9 BackupCell’s Behavior ......................................................... 13
1.10 Postcondition for Method inc .............................................. 13
1.11 Example of an Execution Graph ......................................... 13
2.1 Example of an Execution Path .............................................. 16
2.2 Sort Constructors ............................................................. 20
2.3 Example of Named Rules ..................................................... 21
2.4 Example of Distinguished Elements ...................................... 21
2.5 Example of a Named Tuple ................................................... 21
2.6 Example of a Control Flow Graph ........................................ 22
3.1 Trigger Syntax ................................................................. 25
3.2 Action Syntax ................................................................. 25
3.3 ASM Execution Graph and UML Step .................................... 26
4.1 Collection Operators ......................................................... 39
4.2 $\mu$-Calculus Syntax ......................................................... 42
A.1 Predicate Semantics ......................................................... 48
A.2 Rule Semantics .............................................................. 49
B.1 Method Inheritance Example .............................................. 62
C.1 OCL Expressions Type System .......................................... 78
C.2 Predefined Types for OCL Expressions ............................... 79
C.3 $\mu$-Calculus Semantics ................................................... 79
List of Acronyms and Symbols

Acronyms

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASM</td>
<td>Abstract State Machine</td>
</tr>
<tr>
<td>CTL</td>
<td>Computation Tree Logic</td>
</tr>
<tr>
<td>CRAC</td>
<td>Conception et Réalisation des Applications Complexes</td>
</tr>
<tr>
<td>LTL</td>
<td>Linear Temporal Logic</td>
</tr>
<tr>
<td>OCL</td>
<td>Object Constraint Language</td>
</tr>
<tr>
<td>OMG</td>
<td>Object Management Group</td>
</tr>
<tr>
<td>OMT</td>
<td>Object Modeling Technique</td>
</tr>
<tr>
<td>OOSE</td>
<td>Object-Oriented Software Engineering</td>
</tr>
<tr>
<td>RFP</td>
<td>Request for Proposals</td>
</tr>
<tr>
<td>UML</td>
<td>Unified Modeling Language</td>
</tr>
</tbody>
</table>

Symbols

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Upsilon$</td>
<td>ASM vocabulary</td>
</tr>
<tr>
<td>$A, B, \ldots$</td>
<td>ASM states</td>
</tr>
<tr>
<td>$\bot$</td>
<td>Undefined value</td>
</tr>
<tr>
<td>$\zeta$</td>
<td>Variable assignment</td>
</tr>
<tr>
<td>$\Phi_a, \Phi_b, \ldots$</td>
<td>ASM predicates</td>
</tr>
<tr>
<td>$\Pi_a, \Pi_b, \ldots$</td>
<td>ASM transition rules</td>
</tr>
<tr>
<td>$\mathcal{S}(A)$</td>
<td>Sets of elements of $A$</td>
</tr>
<tr>
<td>Symbol</td>
<td>Definition</td>
</tr>
<tr>
<td>--------</td>
<td>--------------------------------</td>
</tr>
<tr>
<td>\mathcal{L}(A)</td>
<td>Lists of elements of A</td>
</tr>
<tr>
<td>\mathcal{T}(A_1, \ldots, A_n)</td>
<td>n-tuples in $A_1 \times \ldots \times A_n$</td>
</tr>
<tr>
<td>\mathcal{R}(A)</td>
<td>Relations over A</td>
</tr>
<tr>
<td>A_1 \mid \ldots \mid A_n</td>
<td>Elements of $A_1 \cup \ldots \cup A_n$</td>
</tr>
<tr>
<td>A_1 \rightarrow A_2</td>
<td>Finite mappings from $A_1$ to $A_2$</td>
</tr>
<tr>
<td>A^*</td>
<td>Stacks of elements of A</td>
</tr>
<tr>
<td>\emptyset</td>
<td>Empty set</td>
</tr>
<tr>
<td>\langle \rangle</td>
<td>Empty list, empty stack</td>
</tr>
<tr>
<td>\Theta</td>
<td>ASM execution graph</td>
</tr>
<tr>
<td>\rightarrow</td>
<td>ASM transition relation</td>
</tr>
<tr>
<td>\Lambda</td>
<td>UML execution graph</td>
</tr>
<tr>
<td>\longrightarrow</td>
<td>UML step relation</td>
</tr>
<tr>
<td>e \vdash \tau</td>
<td>Expression $e$ is of type $\tau$</td>
</tr>
<tr>
<td>\subseteq</td>
<td>Subtype relation</td>
</tr>
<tr>
<td>\prec_{d}</td>
<td>Direct inheritance relation</td>
</tr>
<tr>
<td>\preceq_{h}</td>
<td>Inheritance relation</td>
</tr>
<tr>
<td>\preceq_{s}</td>
<td>More specific relation</td>
</tr>
<tr>
<td>\preceq_{o}</td>
<td>Overriding relation</td>
</tr>
<tr>
<td>\Theta, \Phi</td>
<td>Observation of $\Theta$ through predicate $\Phi$</td>
</tr>
<tr>
<td>\Theta \models \phi</td>
<td>The $\mu$-formula $\phi$ holds in $\Theta$</td>
</tr>
<tr>
<td>\Lambda \models c</td>
<td>The $\mathcal{OCL}$ constraint $c$ holds in $\Lambda$</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

1.1 Document purpose

The purpose of this document is to present final version of the Socle project semantics of UML and OCL using the ASM formalism.

1.2 About UML

**UML Inception** In the nineteen seventies and eighties, several object-oriented design notations and methodologies were developed. This culminated in the early nineties with over than 50 methodologies competing in what is known as the *method wars*. As these methods matured and inspired each other, the need for an unified notation became clear. In the mid-nineties, Grady Booch and Jim Rumbaugh joined efforts in the fusion of the *Booch* and *OMT* (Object Modeling Technique) methods. Then Ivar Jacobson incorporated his *OOSE* (Object-Oriented Software Engineering) method, which resulted in the Unified Modeling Language. The first specification documents of *UML* (*UML* 0.9) were proposed to the *OMG* (Object Management Group) in 1996. The *OMG* issued an *RFP* (Request for Proposals) as a first step toward a *UML* 1.0 definition. The consortium built around that *RFP* included organizations such as Digital Equipment Corp., HP, i-Logix, IntelliCorp, IBM, ICON Computing, MCI Systemhouse, Microsoft, Oracle, Rational Software, TI and Unisys. *UML* 1.4 is the current version of the standard but the 2.0 version is nearly complete [OMG03a, OMG03b].

**UML Diagrams** The specification of the Unified Modeling Language presents a collection of 12 kinds of diagrams. Structural diagrams allow us to specify the static structure of the application. These include class diagrams, object diagrams, component diagrams and deployment diagrams. Behavioral diagrams represent different aspects of the behavior of the application. They include use case diagrams, sequence diagrams, activity diagrams, collaboration diagrams and statechart diagrams. Model management diagrams are meant to support project management by documenting how development can be broken down into modules and subsystems; they include package diagrams, subsystem diagrams and model diagrams. Rumbaugh gives a complete introduction to *UML* in [RJB98]. In this document, we concentrate on class diagrams, object diagrams and statechart diagrams.

6
CHAPTER 1. INTRODUCTION

Modeling Data Structures in UML  Figure 1.1 shows an example of a class diagram containing every feature covered in this document. A class is the object-oriented mechanism for type definition. A class specification contains both data (fields) and procedures (methods). The object-oriented sub-typing mechanism is called inheritance. For example, class B inherits the fields and methods of class A. Wherever an instance of class A may occur, an instance of class B may also occur. In addition, class B can redefine (override) methods of class A. An association specifies a semantic relationship that can occur between typed instances [OMG03b]. The multiplicity modifier restricts the number of participating instances in that relationship.

Figure 1.1: UML Class Diagram

Modeling Data in UML  Figure 1.2 shows an example of an object diagram containing every feature covered in this document. Objects act both as data store and as context for the execution of methods. Notice how an object is specified by its name, its type and a valuation of its fields. Notice also how field x is inherited from class A. A reference is an instantiated association. The object o1 is allowed to call methods of the objects o1 or o2 through the reference y.

Figure 1.2: UML Object Diagram

Modeling Behavior in UML  Figure 1.3 shows an example of a statechart diagram containing nearly every feature covered in this document. For each class, the UML model must contain a statechart that specifies its behavior. The class control flow is modeled through states and transitions. Note that some states (called composite states) contain sub-states. Instructions for the class methods are modeled by labeling transitions with triggers, guards and actions. A trigger may indicate that a transition is part of a particular method. A guard states a condition on the model configuration that must be met for the transition to be fired. A transition’s actions model instructions such as method calls, field assignments, etc.
Each object of the object diagram has its own state-machine. It consists of a set of active states belonging to the statechart of the object’s class. One step in the model evolution corresponds to the firing of a set of state-machine transitions. As a result, active states are updated by deactivating source states of the fired transitions and activating target states. If a final state is reached, the enclosing state is deactivated, possibly deactivating the state-machine. It can be reactivated by a method call.

1.3 About OCL

The Object Constraint Language was developed at IBM by Jos Warmer as part of the Syntropy method. It was integrated into UML 1.0 and has since gained increasing interest. The purpose of this language is to complement the notation by allowing details that are not easily captured by diagrams to be specified. OCL is easy to understand, yet precise enough for software design. It is divided into OCL expressions and OCL constraints [OMG02]. The latter enable the designer to specify contracts on the intended behavior of the model. The former are used in a UML model wherever an expression is needed.

OCL Expressions

OCL expressions are strongly typed and have no side-effects, i.e. their evaluation does not change the model configuration. Assuming some restrictions on the OCL semantics (see Section ??), an OCL expression evaluates to a precise value given a model configuration. OCL expressions are therefore used to specify values that will change as the model evolves. As such, expressions are useful in a number of places in a UML model:

- To specify the initial value of a field (which may depend on other values)
- To specify the value of a parameter in a method call
- To specify a new value for a field
- To specify a guard in a statechart diagram.
CHAPTER 1. INTRODUCTION

**Evaluation Context**  An OCL expression is always evaluated (and type-checked) according to a context. Statically, the context is a class. During evaluation, that context is an instance of that class. The special variable `self` always refers to that instance. For example, consider the class diagram of Fig. 1.1. Given class `B` as context, the expression “`self.x`” is a legitimate expression. This expression simply accesses the value of field `x`. The expression is of integer type. Evaluated on the object diagram of Fig. 1.2 with the object `o1` as context, it yields the value 0.

**Basic Operators**  Like any programming-language expressions, OCL expressions support literals and basic operators on booleans and integers. Continuing our example, the expression “`self.x < 1`” is a legitimate expression of boolean type. It yields the value `true`.

**Navigation**  The main distinction between OCL expressions and programming language expressions is the support for collections. An OCL expression may navigate object references and collect the resulting set of objects. From a collection, the “.” operator can be reapplied to further navigate the configuration. For example, the expression “`self.y`” is a legitimate expression. The type of the expression is a collection of type `A`. Evaluated on the object diagram of Fig. 1.2, it returns `{o1, o2}`. Further navigation takes the “`self.y.x`” form. Evaluated on our example, it returns `{0, 1}`, i.e. the value of field `x` for the objects `o1` and `o2`.

**Collection Operators**  OCL expressions also include collection operators such as `forall`, `exists`, `filter`, etc. For example, the expression “`self.y.x for all {z | z = 1} `” is a legitimate OCL expression of boolean type. Once evaluated, it returns `false`.

**Constraints**

There are three kinds of OCL constraints: class invariants, method preconditions and method postconditions. A class invariant states that a condition must always be met by all instances of the class. A method precondition states that a condition must be met before the method is invoked. A method postcondition states that a condition must be met after the method has returned. For every kind of constraint, the conditions are specified using an OCL expression of boolean type. Note that a constraint does not modify a model’s behavior. Rather, it is used *a posteriori* to verify whether it satisfies some behavioral requirement.

**Context**  When an OCL expression is used as a guard in a statechart, the context is implicit. In fact, it is always the class associated with the statechart. For OCL constraints, however, the context has to be specified explicitly. For example, Fig. 1.4 shows an invariant for class `B`, stating that it may never reference an object whose field `x` is 10 or higher.

```
context: B
inv: self.y forall{z | z.x < 10}
```

Figure 1.4: Example of an Invariant

For method preconditions and postconditions, the context is a class and a method signature. For example, Fig. 1.5 states that, after every call to method `set`, field `x` must hold the value of the method parameter `i`. 
CHAPTER 1. INTRODUCTION

1.4 Motivating Example

Object-oriented software design is a difficult task, particularly when behavior inheritance is involved. As an illustration, we present the example of a memory cell. Imagine a simple object-oriented component acting as a memory cell. The cell must support updates, incrementations and retrievals of its value. To specify the cell, we create a class `Cell` with the appropriate fields and methods (Fig. 1.6).

```
Cell
content : int
get() : int
set(i: int) : void
inc(i: int) : void
```

Figure 1.6: Example of a Memory Cell

The class behavior is specified by the statechart of Fig. 1.7 (the action %m stands for the return of method m). Notice how method `inc` calls methods `get` and `set` to retrieve and update the cell value. This is good practice, as these methods may display refined behavior in subclasses. Imagine that we want to extend the memory cell specification to include a method that cancels the last update and returns to the previous value. We do that by specifying a new class that inherits from class `Cell`. The new class diagram is shown in Fig. 1.8.

Class `BackupCell` adds field `backup` and method `restore` to the memory cell specification. Its behavior is specified by the statechart diagram of Fig. 1.9. Notice how method `set` is redefined to copy the value of the cell in field `backup`.

Consider that method `inc`, which is not redefined, is called on an instance of class `BackupCell`. The call is answered by the behavior inherited from class `Cell`. Method `get` is called to retrieve the current value. That value is incremented by the integer received as parameter. Then, method `set` is used to set the content of the memory cell to the incremented value. Since method `inc` executes on an instance of class `BackupCell`, the overridden method `set` is called to execute that task. Thus, the behavior defined by the statechart of `BackupCell` is executed and the old content is stored in the `backup` field.

We see how the object-oriented behavior inheritance mechanism can make design difficult. In this example, however, it is quite simple to specify behavioral correctness using an OCL constraint (see Fig. 1.10). It states that, after a call to method `inc`, the `backup` field must hold the value that the `content` field had prior to the method call. This is the meaning of the special `@pre` operator.

In order to verify the constraint, the designer must specify the initial configuration of the model using an
CHAPTER 1. INTRODUCTION

Figure 1.7: Cell’s Behavior

object diagram. From that configuration, the execution graph of the model is computed according to the UML model semantics. During that computation, the OCL expression "self.backup = content@pre" is evaluated over every configuration of the model. The evaluated expression becomes an atomic property, namely \( b_1 \). In addition, configurations are labeled with another atomic property (\( b_2 \)) indicating the return of the inc. Figure 1.11 presents an example of such an execution graph.

The next verification step is to derive a temporal relation from the OCL constraint. In this case, the relation would state that when the model reaches a configuration where \( b_2 \) is true, \( b_1 \) is true. Of course, this relation is formally expressed in a temporal logic. Conceptually, it is verified by searching a path in the graph leading to a configuration where \( b_2 \) and \( \neg b_1 \). If such a path exists, the constraint is violated (as in the case of Fig. 1.11). The designer can then scrutinize the execution graph in order to correct his/her design.

1.5 Document Outline

Abstract State Machine Chapter 2 presents the ASM formalism. The notion of UML model configuration is captured using ASM vocabularies. The notion of UML model step is captured using ASM transition rules. The OCL expression semantics is formalized as an external function. Some details are relegated to Appendix A.

UML Model Semantics Chapter 3 describes how the UML model semantics is developed. Further details are found in Appendix B.

OCL Constraints Semantics Chapter 4 presents how UML model can be model-checked using OCL constraints. Further details are found in Appendix C.
1.6 Glossary

**Field**
A field is a local variable contained in an object. In this document, the word *field* is preferred to *attribute*.

**Method**
A method is a procedure contained in an object. In this document, the word *method* is preferred to *function* or *operation*.

**Object Diagram**
An object diagram is a restricted form of collaboration diagram. Message that objects may exchange are not represented. Its purpose is to specify the initial state of the executable UML model.

**Statechart Diagram**
A statechart diagram specifies the behavior of a class. In this document, *statechart* is used for the diagram specifying the behavior of a class. The actual executable entity (associated to an object) is called *state machine*, i.e. a statechart instance.

**UML Model Configuration**
An UML model configuration is a node in the model execution graph. A configuration contains the active states, the active signals, a set of objects and calling stacks.

**UML Model Step**
An UML model step is an edge in the model execution graph. A step consists of selecting and firing a set of state machine transitions.
CHAPTER 1. INTRODUCTION

Figure 1.9: **BackupCell**’s Behavior

```
context: BackupCell::inc(i:int)
      post: self.backup = (self.content)@pre
```

Figure 1.10: Postcondition for Method **inc**

Figure 1.11: Example of an Execution Graph
Chapter 2

Abstract State Machines

This chapter describes the ASM formalism. An introductory example illustrates important notions: vocabulary, state, term and transition rule. These notions are then defined formally along with the notion of execution graph. We then propose observation operators that we later use to define the UML model and OCL constraints semantics. Finally, we explain how an ASM is specified and enumerate conventions used in the following chapters. Complementary definitions can be found in Appendix A.

2.1 Introductory Example

This section introduces the ASM formalism with a simple example. It models a system where an agent produces messages while another consumes and stores them.

Vocabulary An ASM vocabulary specifies how states are structured. It consists of a collection of sorts and a set of function signatures. Sorts are used to differentiate values. The following sorts identify that there are three kinds of values in our example: agents, messages and message queues:

\[
\text{sort } \text{Agent} \\
\text{sort } \text{Msg} \\
\text{sort } \text{Queue} = \mathcal{L}(\text{Msg})
\]

The latter is a constructed sort, which contains lists of messages. Other constructors include sets, tuples, maps and stacks.

Functions are used to store values within a state. The following functions are used in the example:

\[
\text{static } \text{AGENTS} : S(\text{Agent}) \\
\text{dynamic } q : \text{Queue} \\
\text{dynamic } r : \text{Msg} \rightarrow \text{Bool}
\]

The first two respectively hold the set of agents involved in the example and the message queue used for communication. The third one is used to indicate whether a message is known to the receiver agent.
There are three kinds of functions: static, dynamic and external. Static functions are invariant throughout the evolution of the ASM. Dynamic functions allow the ASM to evolve. For example, the function $q$ will be updated as messages are exchanged. External functions are defined in terms of static and dynamic functions. Hence, their value may evolve as dynamic functions evolve. This mechanism is later used to integrate the evaluation of OCL expressions into the UML model semantics. The evaluation is done with respect to a fixed definition that fetches values in the UML model configuration. External functions are also used to add operators to constructed sorts. For example, the following external functions are used to manipulate message queues:

<table>
<thead>
<tr>
<th>External Function</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{external } _@_ : \text{Queue} \times \text{Queue} \to \text{Queue}$</td>
<td></td>
</tr>
<tr>
<td>$\text{external } \langle _ \rangle : \text{Msg} \to \text{Queue}$</td>
<td></td>
</tr>
<tr>
<td>$\text{external } \text{hd} : \text{Queue} \to \text{Msg}$</td>
<td></td>
</tr>
<tr>
<td>$\text{external } \text{tail} : \text{Queue} \to \text{Queue}$</td>
<td></td>
</tr>
</tbody>
</table>

These functions are assumed to be part of the vocabulary by virtue of the fact that $\text{Queue}$ is a constructed sort. Their interpretation corresponds to the usual interpretation of list operators.

**Initial State** An ASM state defines a carrier set for each sort and assigns an interpretation to each function. The carrier set contains every possible element of a given sort and is infinite in general. A function interpretation is a (potentially) infinite enumeration of the values taken by a function. The initial state of our example is defined as follows:

\[
\begin{align*}
\text{AGENTS} & = \{\text{SND, RCV}\} \\
q & = \langle \_ \rangle \\
r(m) & = \bot \quad \text{for all } m \in \text{Msg}
\end{align*}
\]

The $\bot$ symbol represents the undefined element, which is assumed present for each sort.

**Transition Rule** The evolution of the ASM is specified by the following transition rule:

\[
\begin{align*}
\epsilon_{a \in \text{AGENTS}} : \\
\text{case } a \text{ of} \\
\text{SND} \to \\
\theta_{m: \text{Msg}} : q := q@\langle m \rangle \\
\text{RCV} \to \\
q \neq \langle \_ \rangle \\
? & \quad \| \quad q := \text{tail}(q) \\
& \quad \| \quad r(\text{hd}(q)) := \text{true}
\end{align*}
\]

First, an agent is chosen non-deterministically. If it is the sender agent, a new message is created and appended to the queue. If it is the receiver agent, a message is dequeued (given the queue is not empty). The message becomes known to the receiver as indicated by the update of the function $r$. In general, a rule comprises variables, terms, predicates and subrules.

Variables are created and bound in rules. For example, the rule “$\theta_{m: \text{Msg}}$ :” introduces the variable $m$ and binds it to a fresh message. The variable is available in the subsequent rules and terms. Terms allow functions to be applied. For example, the term “$q@\langle m \rangle$” is the application of the concatenation function to the terms “$q$” and “$\langle m \rangle$”. Predicates are special terms that always evaluate to a boolean value. They are used to parameterize some rules. For example, $q \neq \langle \_ \rangle$ is a predicate that decides if the $?$ clause is evaluated.
Rules serve two purposes. First, the update rule allows us to specify how dynamic functions evolve. For example, 
\[ r(\text{hd}(q)) := \text{true} \] is an update that will assign the boolean value \text{true} to the function \text{r} at location \text{hd}(q).

Other rules are used to model the control flow of the system or to parameterize updates. For example, the choose rule (e.g. \text{e} \in \text{AGENTS}) allows us to make a non-deterministic choice before evaluating the remainder of the rule.

**Execution** From a given state, the next states are computed by evaluating the transition rule. This yields a collection of update sets. Each update describes how a dynamic function will take a new value for a given location. The updates of an update set are applied in no particular order, which means they have to be consistent. For example, \[ q := \langle \rangle \] and \[ q := \langle m \rangle \] would not be allowed in the same update set. Figure 2.1 gives an example of a path in the execution graph.

![Figure 2.1: Example of an Execution Path](image)

First, the transition rule is evaluated in the initial state. The sender agent is selected by the choose rule. The fresh message \text{m} is created by the \text{\@m:Msg} rule and appended to \text{q} by the \text{q := q@hm} update rule. This yields the second state. The transition rule is evaluated again. The receiver agent is selected by the choose rule. As the queue is not empty, the \text{?} clause is evaluated, yielding the following updates: \text{q := tail(q)} and \text{r(m1) := true}. This illustrates how the updates \text{q := tail(q)} and \text{r(hd(q)) := true} are not evaluated in sequence. If they were, the term \text{hd(q)} in the second update would be undefined. The third transition corresponds to the selection of the sender agent again.

### 2.2 State

**Definition 2.1 (ASM Vocabulary)** An ASM vocabulary \( \Upsilon \) is a couple \((S, \Omega)\), where:

- \( S = (S_i)_{i \in I} \) is a collection of sorts
- \( \Omega \) is a set of function signatures of the form \( f : S_1 \times \ldots \times S_k \rightarrow S_n \).

We assume that \( \Omega \) is partitioned into \( \Omega_{\text{stat}}, \Omega_{\text{dyn}} \) and \( \Omega_{\text{ext}} \), respectively static functions, dynamic functions and external functions.

For each sort \( S_i \), we assume a static nullary function \( \bot : S_i \) (interpreted as the undefined element). Finally, we assume a sort \( \text{Bool} \in S \) representing booleans and two static nullary functions \( \text{true} : \text{Bool} \) and \( \text{false} : \text{Bool} \).

**Definition 2.2 (ASM State)** An ASM state \( \mathfrak{A} \) is a couple \((S^\mathfrak{A}, \Omega^\mathfrak{A})\) over a vocabulary \( \Upsilon = (S, \Omega) \), where:

- \( S^\mathfrak{A} = (S_i^\mathfrak{A})_{i \in I} \) is a collection of infinite carrier sets
CHAPTER 2. ABSTRACT STATE MACHINES

- $\Omega^\mathfrak{M}$ is a set of function interpretations of the form $f^\mathfrak{M} \subseteq (S_1^\mathfrak{M} \times \ldots \times S_k^\mathfrak{M}) \times S_n^\mathfrak{M}$.

For each sort $S_i$, we assume that the carrier set $S_i^\mathfrak{M}$ contains the element $\bot^\mathfrak{M}_{S_i}$. The interpretation of $\bot : S_i$ is always $\bot^\mathfrak{M}_{S_i}$. In addition, we assume that true$^\mathfrak{M} \neq$ false$^\mathfrak{M}$.

2.3 Transition

Definition 2.3 (Terms) Consider an ASM vocabulary $\Upsilon = (S, \Omega)$ and a set of variables $v_1 : S_1, \ldots, v_n : S_n$. The set of terms over $\Upsilon$, written $\Delta$, is defined inductively as follows:

- $v_1 : S_1, \ldots, v_k : S_k \in \Delta$
- $f : S_i \in \Omega$, then $f : S_i \in \Delta$
- If $t_1 : S_1, \ldots, t_k : S_k \in \Delta$ and $f : S_1 \times \ldots \times S_k \to S_n \in \Omega$, then
  - $f(t_1, \ldots, t_k) : S_n \in \Delta$.

The evaluation of a term is defined in Section A.1.

Definition 2.4 (Predicates) Consider an ASM vocabulary $\Upsilon = (S, \Omega)$. Let $t$ range over the boolean terms over $\Upsilon$. The set of predicates over $\Upsilon$ (ranged over by $\Phi, \Psi$) is inductively defined according to the following syntax.

$$\Phi ::= t \mid \neg \Phi \mid \Phi \land \Psi \mid \Phi \lor \Psi \mid \Phi \Rightarrow \Psi \mid \Phi \Leftarrow \Psi \mid \Phi \iff \Psi \mid \exists x : S_i \cdot \Phi \mid \forall x : S_i \cdot \Phi$$

The evaluation of predicates is as usual. For definiteness, we give their semantics in Section A.3. It relies on the notion of domain, defined in Section A.2.

Definition 2.5 (Transition Rule) Consider an ASM vocabulary $\Upsilon = (S, \Omega)$. Let $\Phi$ range over the predicates over $\Upsilon$. The set of possible transition rules over $\Upsilon$ (ranged over by $\Pi$) is defined according to the following syntax.

$$\Pi ::= \circ \mid f(t) := s \mid \Phi \land \Pi \mid \Pi \land \Pi \mid \Pi \land \Phi \mid \Pi \land \exists x : S_i \cdot \Phi \mid \Pi \land \forall x : S_i \cdot \Phi \mid x := t : \Pi$$

The informal meaning of the other rules is given below.

Skip $\circ$ No effect
Update $f(t) := s$ Updates the dynamic function $f$ at location $t$ to the value of $s$
Conditional $\Phi \land \Pi_1 \land \Pi_2$ $\Pi_1$ is evaluated if $\Phi$ is true, else $\Pi_2$ is evaluated
Composition $\Pi_1 \land \Pi_2$ Both $\Pi_1$ and $\Pi_2$ are evaluated
$\Sigma$-Composition $\Sigma_{x : S_i} \cdot \Phi \land \Pi$ $\Pi$ is evaluated for every value of $x$ that renders $\Phi$ true
Choose $\epsilon_{x : S_i} \cdot \Phi \land \Pi$ A value for $x$ that renders $\Phi$ true is chosen and $\Pi$ is evaluated
Let $x := s : \Pi$ $x$ is given the value of $s$ and $\Pi$ is evaluated
2.4 Execution Graph

Formally, an Abstract State Machine is defined as follows.

Definition 2.6 (Abstract State Machine) An ASM $M$ is a triple $(\mathcal{V}, \Pi, \mathfrak{A})$, where:

- $\mathcal{V}$ is a vocabulary
- $\Pi$ is a transition rule over $\mathcal{V}$
- $\mathfrak{A}$ is a state over $\mathcal{V}$ (the initial state).

From a given state, the evaluation of the transition rule yields a collection of update sets. The formal rule semantics is explained in Section A.4. It relies on the evaluation of terms given in Section A.1 and the predicate semantics given in Section A.3.

Definition 2.7 (Update Set) Consider an ASM $M = (\mathcal{V}, \Pi, \mathfrak{A})$ and a state over $\mathcal{V}$. The evaluation of $\Pi$ in state $\mathfrak{A}$, written $[\Pi]_{\mathfrak{A}}$, yields a collection $(U)_{i \in I}$ of update sets. Every update set $U_i$ contains elements of the form $(f, (a_1, \ldots, a_k), b)$, where:

- $f : S_1 \times \ldots \times S_k \to S_n$,
- $a_1 : S_1^\mathfrak{A}, \ldots, a_k : S_k^\mathfrak{A}$ and
- $b : S_n^\mathfrak{A}$.

Definition 2.8 (Update Set Consistency) Consider an ASM vocabulary $\mathcal{V}$ and a state $\mathfrak{A}$ over that vocabulary. An update set $U$ is consistent if:

$$\forall u_1, u_2 \in U. \ u_1 = (f, (a_1, \ldots, a_k), b) \land u_2 = (f, (a_1, \ldots, a_k), c) \Rightarrow b = c$$

Definition 2.9 (Applying an Update Set) Consider an ASM vocabulary $\mathcal{V}$ and a state $\mathfrak{A}$ over that vocabulary. Let $U$ be a consistent update set. The result of applying $U$ to $\mathfrak{A}$ is a new state $\mathfrak{B}$ (written $\mathfrak{A} U \rightarrow \mathfrak{B}$), such that:

- For every dynamic function $f \in \Omega$, $f^{\mathfrak{B}}(\bar{a})$ is defined as follows:

$$f^{\mathfrak{B}}(\bar{a}) = \begin{cases} b & \text{if } ((f, \bar{a}), b) \in U \\ f^{\mathfrak{A}}(\bar{a}) & \text{otherwise} \end{cases}$$
For every external function \( f \in \Omega \), \( f^\mathcal{M} \) is consistent with the external definition of \( f \) and the updated interpretations of the dynamic functions of \( \Omega \).

**Definition 2.10 (Execution Graph)** Consider the ASM \( M = (\mathcal{Y}, \Pi, \mathcal{A}) \). Its execution graph \( \Theta_M \) is a triple \((S, \rightarrow, \mathcal{A})\), where \( S \) and \( \rightarrow \) are defined inductively as follows:

- \( \mathcal{A} \in S \)
- If \( \mathfrak{B} \in S \), \( U \in [\Pi]^{\mathfrak{B}} \), \( U \) is consistent and \( \mathfrak{B} \models U \), then:
  - \( \mathcal{C} \in S \)
  - \( (\mathfrak{B}, \mathcal{C}) \in \rightarrow \).

Hence, \( S \) contains the states of the graph, \( \rightarrow \subseteq S \times S \) is the transition relation and \( \mathcal{A} \) is the initial state.

### 2.5 Observations

**State Restriction** We define a state restriction operator that is used to formally define a UML model execution graph in Chapter 3.

**Definition 2.11** Consider a state \( \mathcal{A} = (S, \mathcal{M}) \) over the \( \mathcal{Y} = (S, \Omega) \) vocabulary and another vocabulary \( \mathcal{Y}' = (S', \Omega') \) such that \( S' \subseteq S \) and \( \Omega' \subseteq \Omega \).

The restricted state \( \mathcal{A}|_{\mathcal{Y}'} \) is the pair \((S', \mathcal{M}')\) such that:

- \( S' \mathcal{M}' = (S_i\mathcal{M})_{S_i \in S'} \)
- \( \Omega' \mathcal{M}' = \{f^{\mathcal{M}} \mid f \in \Omega'\} \).

**Graph Observation** We define an execution graph observation criterion to be used in Chapter 4.

**Definition 2.12** Given an execution graph \( \Theta \) and a first-order predicate \( \Phi \) on the vocabulary of \( \Theta \), the observation of \( \Theta \) through \( \Phi \), written \( \Theta |_{\Phi} \), is defined as follows:

- \( \mathcal{A}_0 \rightarrow \mathcal{A}_n \in \Theta |_{\Phi} \iff \) if there is some \( \mathcal{A}_0 \rightarrow \mathcal{A}_1 \rightarrow \ldots \rightarrow \mathcal{A}_n \in \Theta \) such that \( \Phi \) is true in \( \mathcal{A}_0 \) and \( \mathcal{A}_n \) and false in \( \mathcal{A}_i \) for any \( 1 \leq i < n \).

### 2.6 Specifications

This sections lists conventions that we use to specify vocabularies and transition rules throughout this document.
2.6.1 Constructed Sorts

By default, a sort contains simple elements. As illustrated in the example, we allow constructed sorts to be specified. Figure 2.2 lists the constructors considered. Constructed sorts come with the usual functions, defined as external functions. These are listed in Section A.5.

<table>
<thead>
<tr>
<th>Constructor</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S(A)$</td>
<td>Sets of elements of $A$</td>
</tr>
<tr>
<td>$L(A)$</td>
<td>Lists of elements of $A$</td>
</tr>
<tr>
<td>$R(A)$</td>
<td>Relations over elements of $A$</td>
</tr>
<tr>
<td>$T(A_1, \ldots, A_n)$</td>
<td>$n$-tuples in $A_1 \times \ldots \times A_n$</td>
</tr>
<tr>
<td>$A_1 \rightarrow A_2$</td>
<td>Finite mappings from $A_1$ to $A_2$</td>
</tr>
<tr>
<td>$A^*$</td>
<td>Stacks of elements of $A$</td>
</tr>
</tbody>
</table>

Figure 2.2: Sort Constructors

Most of these external functions are conventional. The functions \(\langle\rangle\), \(\oplus\) and \(\text{buildMap}\) ought to be explained. The first fetches the $n^{th}$ element of a list. For example, the term \(\langle a, b, c\rangle.2\) will evaluate to $b$. The second allows us to update a mapping. For example, the term \(\{a \mapsto 1, b \mapsto 2\} \oplus \{b \mapsto 3, c \mapsto 4\}\) will evaluate to \(\{a \mapsto 1, b \mapsto 3, c \mapsto 4\}\). The last one builds a mapping from two lists. For example, the term \(\text{buildMap}(\langle a, b\rangle, \langle 1, 3\rangle)\) will evaluate to \(\{a \mapsto 1, b \mapsto 3\}\).

2.6.2 Syntactic Sugar

Shortcuts

Some shorthand notations are listed below.

<table>
<thead>
<tr>
<th>Long Form</th>
<th>Short Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t = t_1$</td>
<td>$\text{case } t \text{ of }$</td>
</tr>
<tr>
<td>. $\Pi_1$</td>
<td>$t_1 \mapsto \Pi_1$</td>
</tr>
<tr>
<td>( ? ) $\Pi_2$</td>
<td>$t_2 \mapsto \Pi_2$</td>
</tr>
<tr>
<td>( : ) $t = t_3$</td>
<td>$t_3 \mapsto \Pi_3$</td>
</tr>
<tr>
<td>. $\Pi_3$</td>
<td>$\ldots$</td>
</tr>
<tr>
<td>$\epsilon_{x:A}, x \in A : \Pi$</td>
<td>$\epsilon_{x \in F : \Pi}$ with $F : S(A)$</td>
</tr>
<tr>
<td>$\forall x:A, x \in A : \Pi$</td>
<td>$\forall x \in F : \Pi$ with $F : S(A)$</td>
</tr>
<tr>
<td>$\Phi \triangleleft \Pi : \circ$</td>
<td>$\Phi \mapsto \Pi$</td>
</tr>
<tr>
<td>$\exists x_1 : S_1, \Phi_1 \land \exists x_2 : S_2, \Phi_2$</td>
<td>$\epsilon_{x_1 : S_1, x_2 : S_2 : \Phi_2 : \Pi_1}$</td>
</tr>
<tr>
<td>. $\epsilon_{x_1 : S_1, x_2 : S_2 : \Phi_2 : \Pi_1}$</td>
<td>$\epsilon_{x_1 : S_1, x_2 : S_2 : \Phi_2 : \Pi_1}$</td>
</tr>
<tr>
<td>$\Phi_1 : \Phi_2$</td>
<td>$\Phi_1 : \Phi_2$</td>
</tr>
</tbody>
</table>

Named Rules and Predicates

Naming rules and predicates allows us to break long rules into smaller, more readable, rules. For example, the transition rule of the introductory example may be broken up as shown by Fig. 2.3.
Distinguished Elements A sort must sometimes contain distinguished elements bound to static nullary functions. Figure 2.4 gives an example of such a case along with the abbreviated form.

Named Tuples To simplify the manipulation of tuples, we name the positions of a tuple. Figure 2.5 gives an example of a named tuple. Given an element \( t : T(n_1 : A_1, \ldots, n_n : A_n) \), the term \( t.n_i \) extracts the element in the \( i^{th} \) position.

2.6.3 Control Flow Graphs

In Chapter 3 and Appendix B, we use control flow graphs to simplify rule specifications. The approach is similar to Montage [KP97], albeit simpler. For example, imagine we wish to change the control flow of the introductory example to alternate deterministically between the sender and receiver agents. The control graph is depicted in Fig. 2.6 along with the vocabulary extensions it induces. Note that, according to the graph, the initial value of \( ctr \) is SND.

The transitions rule of the example are now as follows:

\[
\begin{align*}
\text{agents} : & \quad \epsilon \\
\text{case } a \text{ of} & \quad \text{snd} \rightarrow \\
\text{snd} : & \quad m : \text{Msg} \rightarrow q := q \oplus \langle m \rangle \\
\text{rcv} : & \quad \Phi_{\text{emptyQueue}}(q) \\
\text{rcv} : & \quad q = \emptyset \\
\text{dequeueMsg} : & \quad \Pi_{\text{dequeueMsg}}(q) \\
\text{dequeueMsg} : & \quad \text{head}(q) = \text{true} \\
\text{Fig. 2.3: Example of Named Rules}
\end{align*}
\]
Figure 2.6: Example of a Control Flow Graph

The main rule $\Pi$ and subrules $\Pi_{toS}$ and $\Pi_{toR}$ are induced from the graph. Note how the rule $\Pi_{toR}$ calls the user-defined rule $\Pi^*_{toR}$. This allows us to add user-defined behavior to transitions. In our example, we use this mechanism to model the fact that a message may be lost.
Chapter 3

UML Model Semantics

This chapter presents a high-level description of the formal UML model semantics. We begin by detailing the formalization rationales and completing the informal description of the UML fragment considered. We explain how we use the ASM formalism to capture the concepts of configuration and step. We sketch the formalization of a model’s static semantics. We explain how behavior inheritance is captured. We detail the semantics of actions. Finally, we give a formal definition of a UML model execution graph and define safety properties for the semantics. The complete list of formal definitions is given in Appendix B.

3.1 Formalization Rationales

As explained in [OMG03b], the UML statechart semantics is meant to be an object-oriented extension of Harel’s statechart semantics. We formalized statechart semantics by directly adapting a fragment of Harel’s statechart semantics, as given in [HN96]. The object-oriented extensions are supported by selecting a fragment of the triggers and actions proposed in [OMG03b]. The static semantics is directly inspired by the ASM Java semantics of Stark et al. [SSB01].

Statechart The notion of or-states and and-states is supported. We also adopt the idea that transitions effects are applied all at once. We formalize the notion of transition conflict: two transitions are in conflict when they cannot be fired in the same step. The notion of transition priority is dropped. We extend the notion of conflict to include a dynamic computation of conflicts. For example, transitions resulting in incoherent assignation to a field are detected as conflictual.

We apply some simplifications. We do not consider compound transitions. In particular, this means that a transition exiting the initial state of a composite state is not fired in the same step as the transition activating its parent. Instead of pseudo states, initial states are considered as normal states. They are automatically activated when the direct parent state is activated. We do not consider actions associated with entering or exiting a state. We do not consider actions associated with a state being active. Finally, history states are dropped.

Classes We consider associations between classes as a field of the client class. The type of the field is always a list of the supplier class. Multiplicity is supported, but no other special meaning of the association relation is taken
into account, e.g. agglomerates are not supported. Finally, we do not consider interfaces.

**Methods** Since transitions are fired in no particular order, method call actions are always in conflict. Otherwise, firing two transitions may result in an inconsistent update of a calling stack. Consequently, there is at most one method call per step.

**Threads** Concurrency is modelled through a multi-threaded scheduler. The current thread is chosen non-deterministically at the beginning of the step computation. The current thread must be the same throughout the step computation since OCL expressions are evaluated according to an environment containing local variables, i.e. parameters of the currently executing method.

### 3.2 Considered Fragment

As explained in the introduction, we consider a UML model to be exactly one class diagram, one statechart diagram for each class and one object diagram.

**Configuration** A configuration of the model consists of a set of active signals and a set of objects. Each object is specified by its class and the values taken by its fields. If the object is a thread, it also possesses a calling stack. Finally, each object has a state-machine.

**Step** One step of the UML model is the firing of a set of state-machine transitions. The active states of an object’s state-machine denote this object’s local configuration. States allow us to model the basic control flow of the object. Actions are equivalent to instructions that the object can execute. Triggers are used to restrict when such instructions are executed, e.g. by which method.

**States** There are five kinds of states. Initial states are automatically activated when their parent state is entered. Final states automatically deactivate their parent state when entered. Simple states have no special meaning. Finally, there are two kinds of composite states: OR-states, which may contain only one active substate and AND-states, which may contain only OR-states as substates. As long as the AND-state is active, every enclosed OR-state is also active. As soon as one enclosed OR-state is deactivated, the AND-state is also deactivated.

**Transitions** For a transition to be enabled, its source state must be active. The set of states deactivated and activated by the firing of a transition is statically computed. The definition depends on the notion of a transition scope. The scope of a transition is the lowest state (in the state hierarchy) that contains both the source state and the target state of the transition. When a transition is fired, the scope state is deactivated along with every state it encloses. Then, the scope state is reactivated along with the target state. The only exception is when the target state is a final state.

**Triggers** For a transition to be enabled, its trigger must correspond to an active event of the configuration. Figure 3.1 presents the possible triggers of the considered fragment.
The empty trigger $\texttt{*}$ is always satisfied. A signal trigger is satisfied if the corresponding signal is active in the configuration. A method call trigger is satisfied if that method is the currently executing method, i.e., if the method is on top of the current thread calling stack. A method return trigger is satisfied if the method was the last to return.

**Guards** A guard is an OCL expression of boolean type. Section 4.1 gives the formal syntax and semantics of OCL expressions. The guard is evaluated on the current configuration and must evaluate to true for the transition to be enabled.

**Actions** Figure 3.2 gives the syntax of actions. A transition contains a list of actions. When a transition is fired, the list is iterated and the effect of every action is computed. A signal action activates the corresponding signal in the next configuration. An assignment action updates the value of a field. An object creation action adds a new object to a reference field. A new state-machine is created for that object. An object deletion action removes one object from a reference field. Another way of removing objects from a reference field is to use an assignment action, which reassigns the field to a subset of the referenced objects. A method call action places a new call on the current thread’s calling stack. A method return action removes the current method from the current stack.

Notice how the method signature is not given explicitly in method calls. Rather, it is elaborated according to the type of the OCL expressions. This is explained in Section B.1.4. Furthermore, there are two kinds of method calls: one calls the first applicable method according to the inheritance relation, while the second calls the method of the superclass. This is part of the object-oriented behavioral inheritance mechanism.
3.3 Using the ASM Formalism

The static structure and the dynamic configuration of a UML model are captured by different ASM vocabularies, respectively $\mathcal{Y}_{\text{STAT}}$ and $\mathcal{Y}_{\text{UML}}$. A third vocabulary, namely $\mathcal{Y}_{\text{AUX}}$, captures auxiliary functions. The step semantics is captured by an ASM transition rule, namely $\Pi_{\text{UML}}$.

Formally, a UML model configuration is an ASM state over the $\mathcal{Y}_{\text{UML}}$ vocabulary. In order to compute the set of the next configurations, that state is extended with states over the $\mathcal{Y}_{\text{STAT}}$ and $\mathcal{Y}_{\text{AUX}}$ vocabularies. An ASM execution graph is computed from that extended state using the $\Pi_{\text{UML}}$ rule. The final states of that graph contain the next configurations. Each such state is restricted to the $\mathcal{Y}_{\text{UML}}$ vocabulary in order to extract the configuration. Transitions corresponding to the computed steps are added to the UML model execution graph. Figure 3.3 illustrates the approach. The process is iterated in order to compute the entire UML model execution graph. The formal definition of a UML model execution graph is given in Section 3.8.

The $\mathcal{Y}_{\text{STAT}}$ vocabulary contains static information about the UML model. This includes the set of classes, the inheritance relation, the set of method signatures, etc. The $\mathcal{Y}_{\text{UML}}$ vocabulary contains the UML model configuration itself. This includes objects, method instances, active states, etc. The $\mathcal{Y}_{\text{AUX}}$ vocabulary contains auxiliary functions, e.g. functions to iterate through objects and transitions.

For each of these vocabularies, we define well-formedness constraints on the corresponding ASM states. A well-formedness constraint for $\mathcal{Y}_{\text{STAT}}$ corresponds to a syntactic correctness property. It insures that the UML model is well-formed, that statecharts are valid, that types are coherent, etc. A well-formedness constraint for $\mathcal{Y}_{\text{UML}}$ refers to a valid UML configuration. For example, it insures that values are of the right type. Another type of well-formedness property for the $\mathcal{Y}_{\text{UML}}$ vocabulary is constraints on the initial configuration of the model. A well-formedness property for $\mathcal{Y}_{\text{AUX}}$ corresponds to the initial values that the auxiliary functions must take.

The static semantics is captured with the constraints on the $\mathcal{Y}_{\text{STAT}}$ vocabulary and on the initial configuration. The constraints on the $\mathcal{Y}_{\text{UML}}$ vocabulary capture the idea of type safety (see Section 3.9). The next three sections outline the vocabularies of the formal UML model semantics.
The $\Upsilon_{\text{stat}}$ defines a data structure to which diagrams are compiled. It consists of a direct representation of the diagrams’ syntax and a set of elaborated functions. The complete specification of the $\Upsilon_{\text{stat}}$ vocabulary can be found in Section B.1.

3.4.1 Diagram Information

The UML model diagrams are captured using a set of simple and constructed sorts. Classes, objects and signals are simple sorts:

- sort $\text{Class}$
- sort $\text{Obj}$
- sort $\text{Signal}$

States and transitions are captured by constructed sorts:

- sort $\text{StateName}$
- sort $\text{StateType} = \{\text{INIT, FINAL, SIMPLE, OR, AND}\}$
- sort $\text{State} = T(n : \text{StateName}, ty : \text{StateType})$
- sort $\text{Trans} = T(n : \text{TransName}, src : \text{State}, \ldots, tgt : \text{State})$

Fields are captured as follows:

- sort $\text{FieldName}$
- sort $\text{Field} = T(n : \text{Fieldname}, ty : \text{Type}, \ldots)$
- sort $\text{ClassField} = T(c : \text{Class}, f : \text{Field})$

Method signatures are captured as follows:

- sort $\text{MethName}$
- sort $\text{ParamName}$
- sort $\text{Param} = T(n : \text{ParamName}, ty : \text{Type})$
- sort $\text{MSig} = T(\text{meth} : \text{MethName}, \text{args} : L(\text{Param}), \ldots)$
- sort $\text{CMSig} = T(c : \text{Class}, msig : \text{MSig})$

Triggers are captured as follows:

- sort $\text{TriggerType} = \{\text{SIGNAL, METHOD} \ldots\}$
- sort $\text{TSignal} = T(s : \text{Signal}, ty : \text{TriggerType})$
- sort $\text{TMsig} = T(msig : \text{MSig}, ty : \text{TriggerType})$

Actions are captured as follows:
CHAPTER 3. UML MODEL SEMANTICS

sort \( \text{ActionType} \) ≡ \{\text{SIGNAL, MCALL} \ldots\}  
sort \( \text{CallType} \) ≡ \{\text{EXPLICIT, VIRTUAL, SUPER}\}  
sort \( \text{ASignal} \) = \( T(s : \text{Signal}, ty : \text{ActionType}) \)  
sort \( \text{AMCall} \) = \( T(\text{rev} : \text{OclExp}, c : \text{Class}, msig : Msig, \ cty : \text{CallType}, \text{args} : L(\text{OclExp}), ty : \text{ActionType}) \)  

[\ldots]  
sort \( \text{Action} \) = \( \text{ASignal} | \text{AMsig} | \ldots \)

Static functions are used to actually store the diagrams. For example, the following functions hold the statechart diagrams of a UML model:

static \( \text{STATES} \) : \( \text{Class} \rightarrow S(\text{State}) \)  
static \( \text{TRANS} \) : \( S(\text{Trans}) \)  
static \( \text{TOP} \) : \( \text{Class} \rightarrow \text{State} \)  
static \( \text{CHILD} \) : \( \text{State} \rightarrow S(\text{State}) \)  
static \( \text{UP} \) : \( \text{State} \rightarrow \text{State} \)

3.4.2 Elaborated Information

Some important information is elaborated and stored in static functions. For example, the following functions capture elaborated information about statechart diagrams:

static \( \text{CHILD}^* \) : \( \text{State} \rightarrow S(\text{State}) \)  
static \( \text{DEACTIVATES} \) : \( \text{Trans} \rightarrow S(\text{State}) \)  
static \( \text{ACTIVATES} \) : \( \text{Trans} \rightarrow S(\text{State}) \)  
static \( \text{INITSTATES} \) : \( \text{Class} \rightarrow S(\text{State}) \)

The function \( \text{CHILD}^* \) returns all the children of a state (i.e. \( \text{CHILD}^* \) is the transitive closure of \( \text{CHILD} \)). The functions \( \text{DEACTIVATES} \) and \( \text{ACTIVATES} \) indicate which states are deactivated or activated by a transition, respectively. The function \( \text{INITSTATES} \) indicates which states should be initially active.

The following functions are elaborated from the information of the class diagram:

static \( \preceq_h \) : \( \mathcal{R}(\text{Class}) \)  
static \( \text{FIELDS}^* \) : \( \text{Class} \rightarrow S(\text{ClassField}) \)  
static \( \text{METHOD}^* \) : \( \text{Class} \rightarrow S(\text{MSig}) \)  
static \( \preceq_o \) : \( \mathcal{R}(\text{ClassMSig}) \)  
static \( \text{LOOKUP} \) : \( \text{Class} \times \text{ClassMSig} \rightarrow \text{Class} \)

The inheritance relation is computed from the direct inheritance relation present in the model. The \( \preceq_o \) relation indicates whether a method overrides one of the method of its superclass. The static function \( \text{FIELDS}^* \) returns every field of a class, including inherited fields. It is defined as follows:

Elaborated Function 3.1 (Fields of a Class) Given a class \( A \), \( \text{FIELDS}^*(A) \) is defined as follows:

\[
\text{FIELDS}^*(A) = \{(B, f) \mid A \preceq_h B \land f \in \text{FIELDS}(B)\}
\]
3.4.3 Well-formedness

We give two examples of well-formedness properties on states over the $\mathcal{Y}_{stat}$ vocabulary. The first property states that an OR-state may have only one initial state.

**Well-formed Model 3.1 (OR-state Structure)** An OR-state $s$ is well-formed if:

$$\exists s' \in (\text{CHILD}(s) \setminus \{s\}) . s'.ty = \text{INIT}$$

The second property states that an AND-state may have only OR-states as direct children.

**Well-formed Model 3.2 (AND-state Structure)** An AND-state $s$ is well-formed if:

$$\left(\text{CHILD}(s) \setminus \{s\}\right) \neq \emptyset \quad \text{and} \quad \forall s' \in (\text{CHILD}(s) \setminus \{s\}) . s'.ty = \text{OR}$$

The complete list of well-formedness constraints over the $\mathcal{Y}_{stat}$ vocabulary can be found in Section B.1.3.

3.5 $\mathcal{Y}_{\text{UML}}$

The $\mathcal{Y}_{\text{UML}}$ vocabulary holds the model configuration. It is also a direct representation of the model’s object diagram. A complete specification of the vocabulary can be found in Section B.2.1.

**State-machines** An UML model configuration contains active states for each state-machine and the set of active signals:

- **dynamic signals**: $\mathcal{S}(\text{Signal})$
- **dynamic actstates**: $\text{Obj} \to \mathcal{S}(\text{State})$

**Objects** In order to represent objects, we need a sort to specify the values that a field may take. A value is either a boolean, an integer, an object or a list of these simple values:

- **sort Int** $\equiv \{\ldots, -1, 0, 1, \ldots\}$
- **sort Bool** $\equiv \{\text{true}, \text{false}\}$
- **sort SimpleVal** $= \text{Int} \mid \text{Bool} \mid \text{Obj}$
- **sort ListVal** $= \mathcal{L}(\text{SimpleVal})$
- **sort Val** $= \text{SimpleVal} \mid \text{ListVal}$

The function $\text{heap}$ holds the configuration’s objects. An object is a couple containing its class and the values taken by its fields:
CHAPTER 3. UML MODEL SEMANTICS

sort \textit{ObjEnv} = \textit{ClassField} \rightarrow \textit{Val}

sort \textit{Heap} = T(c : \textit{Class}, env : \textit{ObjEnv})

dynamic \textit{heap} : \textit{Obj} \rightarrow \textit{Heap}

**Methods** The function \textit{cont} (shorthand for \textit{execution context}) holds a calling stack for every thread of the configuration. Each frame of the stack contains the current method, a local variable assignation, the sender object and the receiver object:

sort \textit{FrameEnv} = \textit{Var} \rightarrow \textit{Val}

sort \textit{Frame} = T(mth : \textit{ClassMSig}, env : \textit{FrameEnv}, snd : \textit{Obj}, rcv : \textit{Obj})

sort \textit{CallStack} = \textit{Frame}^*

dynamic \textit{cont} : \textit{Obj} \rightarrow \textit{CallStack}

dynamic \textit{returns} : \textit{Obj} \rightarrow \textit{MSig}

3.5.1 Well-formedness

We give an example of a well-formedness constraint over the $\mathcal{U}_{\text{uml}}$ vocabulary.

**Well-formed Configuration 3.1 (Environment)** For every object $o : \textit{Obj}$ such that $\textit{heap}(o) \neq \perp$. Let $C$ be that object’s class (i.e. $C = \textit{heap}(o).c$). Then, we must have:

$$\forall (D, f) \in \text{FIELDS}^*(C) \cdot \textit{heap}(o).env((D, f)) \neq \perp$$

It states that every field of an object, including inherited fields, has a value. The complete list of well-formedness constraints on the $\mathcal{U}_{\text{uml}}$ vocabulary can be found in Section B.2.2.

3.6 $\mathcal{Y}_{\text{AUX}}$

Additional vocabulary is necessary to capture the step semantics. The complete specification of the vocabulary can be found in Section B.3. We need sets containing enabled transitions and transitions to be fired. In addition, we use a boolean nullary function to indicate that the step computation is finished:

dynamic \textit{enabledSet} : $\mathcal{S}(T(\textit{Obj}, \textit{Trans}))$

dynamic \textit{fireSet} : $\mathcal{S}(T(\textit{Obj}, \textit{Trans}))$

dynamic \textit{step} : \textit{Bool}

We also need functions to hold the current thread, current object, etc.:

dynamic \textit{\tilde{\text{th}}} : \textit{Obj} \quad \text{Current thread}

dynamic \textit{\tilde{o}} : \textit{Obj} \quad \text{Current object}

dynamic \textit{\tilde{t}} : \textit{Trans} \quad \text{Current transition}

dynamic \textit{\tilde{a}} : \mathcal{L}(\textit{Action}) \quad \text{Current actions}

In addition, an external function is defined in Chapter 4 for evaluating \textit{OCL} expressions:
The function maps an OCL expression and a variable environment to a value. The variable environment contains the variable `self`, which is mapped to the current object and the local variables located on the top of the calling stack.

Finally, we need to hold a temporary configuration. For each function in the $\UML$ vocabulary, we require that the $\UML_{aux}$ vocabulary contain a counterpart. For example, the function $\text{heap}$ is defined in the $\UML_{aux}$ vocabulary and is required to hold the same information as the function $\text{heap}$ at the beginning of the step computation.

### 3.6.1 Well-formedness

We give two examples of well-formedness constraints on the $\UML_{aux}$ vocabulary. Both define the value that a function must take at the beginning of the step computation.

**Additional Vocabulary 3.1 (Enabled Set)**

$$\text{enabledSet} = \emptyset$$

**Additional Vocabulary 3.2 (Fire Set)**

$$\text{fireSet} = \emptyset$$

The complete specification of well-formedness constraints of the $\UML_{aux}$ vocabulary is given in Section B.3.1.

### 3.7 Step

The basic step semantics is to accumulate enabled transitions and fire them all at once. As some transitions may be in conflict, a subset of non-conflicting transitions is non-deterministically chosen. For a transition to be enabled, its actions must be allowed in the current configuration. Similarly, transitions are conflicting if their actions are in conflict.

The ASM rule $\Pi_{\UML}$ capturing the step semantics has the following control flow:
The next section describes the role of each subrule. Section 3.7.2 explains how object-oriented behavior inheritance is formalized. Section 3.7.3 details the semantics of actions. The complete specification of the transition rule is given in Section B.4.

### 3.7.1 Step Computation

**Choosing a Thread** The first subrule chooses the current thread of execution. It models the controller of a thread-based concurrency scheme. The choice is made non-deterministically as follows:

\[
\begin{align*}
\epsilon_{\omega:Obj} \cdot heap(o).c \in \text{THREADS} : \\
\parallel \quad th := o
\end{align*}
\]

**Collecting Enabled Transitions** The second subrule collects enabled transitions by iterating objects and transitions:

A transition is selected if it is enabled with respect to its source state and trigger. The rule \( \Pi_{\text{evalTransition}} \) determines whether the transition is enabled with respect to its actions. A signal action can be fired in any configuration. For other actions, we require that the current object \( th \) be on top of the current calling stack (as the receiver object). This is necessary because these actions will update either heap or cont. Intuitively, these actions are instructions of the currently executing method. Finally, object creation, object deletion and assignment actions must not violate the multiplicity of the field they update.

**Selecting Transitions to Fire** The third subrule consists of the non-deterministic choice of a (maximal) set of non-confllicting transitions. The fact that transitions are chosen non-deterministically guarantees that every possible set is considered. To guarantee that the set is maximal, we require that the rule \( \Pi_{\text{FT}} \) be called only when no more transitions can be added:
A transition is selected for addition only if it is not in conflict with a previously added transition (with respect to the states it deactivates). The rule \( \Pi_{\text{evalConflict}} \) determines whether a transition can be added with respect to its actions. This allows us to enforce assignation coherence and, also, the fact that only one method call may occur in a step.

**Firing Transitions** The fourth subrule fires each transition:

When a transition is fired, its actions are iterated and fired according to the action semantics of Section 3.7.3. In addition, given a transition \( t \) belonging to the state-machine of object \( o \), the active states are updated as follows:

\[
\overrightarrow{\text{actstates}}(o) := (\overrightarrow{\text{actstates}}(o) \setminus \text{DEACTIVATES}(t)) \cup \text{ACTIVATES}(t)
\]

**Applying Updates** Once every transition is fired, the next configuration is computed. This includes applying updates and garbaging objects:
Updates are applied according to the following rule:

\[
\begin{align*}
& \parallel \signals := \overline{\signals} \\
& \parallel \Sigma_o : \actstates(o) := \overline{\actstates(o)} \\
& \parallel \Sigma_o : \heap(o) := \overline{\heap(o)} \\
& \parallel \cont(\th) := \overline{\cont(\th)} \\
& \parallel \returns(\th) := \overline{\returns(\th)}
\end{align*}
\]

Objects deleted during step computation are garbaged as follows:

\[
\parallel \Sigma_o : \neg \Phi_{\text{referencedObj}}(o) : \heap(o) := \bot
\]

The \(\Phi_{\text{referencedObj}}\) predicate indicates whether a reference to an object exists, either in the heap or in a calling stack. Finally, rule \(\Pi_{\text{done}}\) indicates that the step computation is finished by setting \(\text{step}\) to \(\text{true}\).

### 3.7.2 Behavior Inheritance

With the appropriate static semantics, formalizing behavioral inheritance is straightforward. First of all, the state-machine of an object is the concatenation of its class statechart and its superclasses statecharts. This allows the object to execute inherited behavior. This is achieved by correctly defining the function \(\text{INITSTATES}\).

We then define a mechanism to decide whether an object executes its own behavior or an inherited behavior. According to object-orientation, this is done when a method call is fired. The function \(\text{LOOKUP}\) is used to select which class defines the behavior that should answer the call. That class is either the receiving object class or a superclass of the latter. We make sure the inherited behavior is ready to answer the call by reactivating the initial states using the static function \(\text{INITSTATES}\).

Subsequently, we add the following condition for a transition \(t\) of object \(o\) to be enabled:
\[
\text{top}(cont(\hat{t})).rcv = o \\
\land \ t.src \in \text{STATES}(\text{top}(cont(\hat{t}))).\text{cmsig}.c
\]

The predicate states that the transition is enabled if the object is currently executing. Furthermore, it verifies that the transition corresponds to a behavior inherited from the class selected to answer the call.

### 3.7.3 Action Semantics

Given an action \( a \), its effects are computed according to its kind.

**Signal Action** The signal is simply added to the set of active signals:

\[
\overline{\text{signals}} := \overline{\text{signals}} \cup \{ a.s \}
\]

**Assignation Action** The field environment of the corresponding object is updated:

\[
\begin{align*}
C & := \text{heap}(\hat{o}).c : \\
\text{env} & := \text{heap}(\hat{o}).\text{env} : \\
\text{val} & := [a.\text{exp}]_{\zeta} : \\
\text{heap}(\hat{o}) & := (C, \text{env} \oplus \{ a.cf \mapsto \text{val} \})
\end{align*}
\]

Note that \( \text{exp} \) is the OCL expression of the assignation action and \( \zeta \) is the variable environment needed to evaluate the expression.

**Object Creation** The object creation action is fired as follows:

\[
\partial_{\hat{o} : \text{Obj}} : \\
C := a.cf.f.t : \\
\| \text{env} := \text{heap}(\hat{o}).\text{env} : \\
\text{val} := o^\prime :: \text{env}(a.cf) : \\
\text{heap}(\hat{o}) := (C, \text{env} \oplus \{ a.cf \mapsto \text{val} \}) \\
\| \text{heap}(o^\prime) := \text{NEWOBJ}(C) \\
\| \text{actstates}(o^\prime) := \text{INITSTATES}(C) \\
\| \Pi_{\text{createThread}}(C, o^\prime)
\]
First, a new element is added to the sort \textit{Obj} using the sort extension mechanism of the \textit{ASM} formalism. The new object is added to the current object’s appropriate field. The new object is created and added to function heap. The static function \texttt{NEWOBJ} returns an object with default values for each field. Its initial states are added to function \textit{actstates}. Finally, the rule $\Pi_{\text{createThread}}(C,o')$ verifies whether the new object is a thread and, if so, correctly updates the function \textit{cont}.

**Object Deletion** Executing an object deletion action is straightforward. An object is selected from the object reference field specified in the object deletion action. The reference is removed from the list and the object’s field environment is updated. If it is the last reference to that object, it will be garbaged when updates are applied.

**Method Call Action** Firing a method call action is done as follows. First, the receiver object \textit{rcv} and the method parameters \textit{vals} are computed using the OCL expression evaluation function. If it is a virtual method call, the class \textit{C}, which defines the behavior that answers the call, is $\text{LOOKUP}(\text{heap}(\textit{rcv}).c,a.cmsig)$. If it is a super method call, that class \textit{C} is $\text{LOOKUP}(	ext{SUPER}(\text{heap}(\textit{rcv}).c),a.cmsig)$. Once this information is computed, the action is fired according to the following rule:

\[
\begin{align*}
\text{actstates}(\bar{o}) & := \text{actstates}(\bar{o}) \cup \text{INITSTATES}(C) \\
\text{env} & := \text{buildMap}(\text{msig}:p, \text{vals}) : \\
\text{cont}(\bar{th}) & := \text{push}(\text{cont}(\bar{th}), ((C,a.cmsig.msig), \text{env}, \bar{o}, \textit{rcv}))
\end{align*}
\]

It reactivates the initial states corresponding to the statechart that should answer the call. It pushes the method on the current thread calling stack. The external function \texttt{buildMap} builds the frame environment given a list of parameter names and a list of values.

**Method Return Action** If the action is a method return, the current executing method is popped. In addition, the returned value is stored. This is done by the following rule:

\[
\begin{align*}
\text{cont}(\bar{th}) & := \text{pop}(\text{cont}(\bar{th})) \\
\text{returns}(\bar{th}) & := (a.msig, [a.exp]_\zeta)
\end{align*}
\]

### 3.8 Execution Graph

Assuming the vocabularies $\Upsilon_{\text{STAT}}$, $\Upsilon_{\text{UML}}$, $\Upsilon_{\text{AUX}}$, well-formedness properties for each vocabulary and the \textit{ASM} transition rule $\Pi_{\text{UML}}$, we formally define the notion of \textit{UML} step. The relation $\rightarrow^*$ is the transitive closure of the \textit{ASM} transition relation. The \textit{ASM} state $\mathcal{E}|_{\Upsilon_{\text{UML}}}$ is the restriction of the state $\mathcal{E}$ to the $\Upsilon_{\text{UML}}$ vocabulary (see Section 2.5).

**Definition 3.1 (\textit{UML} Step)** Let $\mathfrak{A}, \mathfrak{A}'$ be two \textit{ASM} states over the $\Upsilon_{\text{UML}}$ vocabulary. Let $\mathfrak{B}$ be a well-formed \textit{ASM}
state over $\Upsilon_{\text{STAT}}$. Let $C$ be a well-formed ASM state over $\Upsilon_{\text{AUX}}$. Let $\Pi_{\text{UML}}$ be the ASM rule capturing the step semantics of UML and let $\rightarrow$ be the transition relation defined by $\Pi_{\text{UML}}$ over $\Upsilon_{\text{STAT}} \cup \Upsilon_{\text{AUX}} \cup \Upsilon_{\text{UML}}$.

There is a UML step from $A$ to $A'$ (written $A \rightarrow A'$) iff there exists an ASM state $E$ over $\Upsilon_{\text{STAT}} \cup \Upsilon_{\text{AUX}} \cup \Upsilon_{\text{UML}}$ such that:

1. $(A \cup B \cup C) \rightarrow^* E$
2. $\text{step}^E = \text{true}$
3. $A' = E|_{\text{UML}}$.

Using that definition, we define the notion of UML execution graph.

**Definition 3.2 (UML Execution Graph)** An UML execution graph $\Lambda$ is a triple $(U, A_0, \rightarrow)$, where:

1. $U$ is a set of configurations
2. $A_0 \in U$ is the initial, well-formed, configuration
3. $\rightarrow \subseteq U \times U$ is the step relation.

### 3.9 Safety Properties

The correctness of an operational semantics is usually proved with respect to a formal declarative description of the semantics. This is not possible for a UML semantics since the declarative description is informal. However, we will define two correctness properties: execution safety and type safety. The first one guarantees that the computation of a step always finishes. The second one guarantees that dynamic values are always of the correct type. Proving these properties is beyond the scope of this document.

**Definition 3.3 (Execution Safety)** Let $A$ be a well-formed UML model configuration. Let $B$ be a well-formed ASM state over $\Upsilon_{\text{STAT}}$. Let $C$ be a well-formed ASM state over $\Upsilon_{\text{AUX}}$. Let $\Pi_{\text{UML}}$ be the ASM rule capturing the step semantics of UML.

The UML model semantics is execution-safe if the ASM graph of $(S, \Pi_{\text{UML}}, A \cup B \cup C)$ is finite and acyclic.

**Definition 3.4 (Type Safety)** Let $A$ be a well-formed UML model configuration.

The UML model semantics is type-safe if every configuration $B'$ such that $B \rightarrow B'$ is a well-formed configuration.
Chapter 4

OCL Semantics

This chapter presents the semantics of OCL expressions and OCL constraints. We define the syntax of OCL expressions and develop their semantics as a recursive function. The function requires a variable assignment and fetches values in the UML model configuration. We give the formal syntax of OCL constraints in terms of an adapted μ-calculus. We explain how it relies on extensions to the Υ_{uml} vocabulary. We show how class invariants, method preconditions and method postconditions are captured as templates. We give a formal semantics to OCL constraints in terms of the adapted μ-calculus semantics. Finally, we explain how the atomic properties of a constraint are evaluated during the computation of the UML model execution graph. Some details are relegated to Appendix C.

4.1 Expressions

4.1.1 Syntax

Sorts OCL expressions are captured by the following sorts, some of which have already been introduced:

sort \textit{Val}
sort \textit{Var}
sort \textit{OclOp} \equiv \{., \textit{iterate}, \textit{@pre}\}
sort \textit{Uop}
sort \textit{Bop}
sort \textit{ClassField}
sort \textit{OclExp}

Variables include method parameters, OCL variables appearing in collection operators and the special variable \textit{self}. We consider three OCL operators: the navigation operator ".", the collection operator \textit{iterate} and the special \textit{@pre} operator. The sorts \textit{Bop} and \textit{Uop} represent the usual binary and unary operators on integers, booleans and lists. Figure C.2 in Appendix C gives the list of the considered operators along with their predefined types. Finally, the sort \textit{OclExp} contains elements constructed according to the OCL expression syntax.

Definition 4.1 (OCL Expression Syntax) Let \(v\) range over \textit{Val}, \(x\) range over \textit{Var}, \(\Delta\) range over \textit{Bop} and \textit{Uop}, \(cf\) range over \textit{ClassField} and \(e\) range over \textit{OclExp}. An OCL expression \(e : OclExp\) must satisfy the following syntax.

38
Chapter 4. OCL Semantics

\[ e ::= v \mid x \mid e \land e \mid e \lor e \mid e \cdot cf \mid e \cdot iterate(x ; x = e \mid e) \mid e @pre \]

Figure 4.1 shows how collection operators are encoded using the \texttt{iterate} operator. Note that \texttt{"\mid"} is the boolean OR operator, while \texttt{"\lor\mid"} is its exclusive XOR counterpart.

<table>
<thead>
<tr>
<th>Function</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{size}</td>
<td>\texttt{iterate(v1; v2 = 0</td>
</tr>
<tr>
<td>\texttt{forall}</td>
<td>\texttt{iterate(v1; v2 = true</td>
</tr>
<tr>
<td>\texttt{exists}</td>
<td>\texttt{iterate(v1; v2 = false</td>
</tr>
<tr>
<td>\texttt{unique}</td>
<td>\texttt{iterate(v1; v2 = false</td>
</tr>
</tbody>
</table>

Figure 4.1: Collection Operators

Functions The OCL expression semantics is integrated into the UML model semantics by means of four external functions:

\[
\begin{align*}
\text{sort} & : \text{OclEnv} = \text{Var} \rightarrow \text{Val} \\
\text{external} & : \text{OclExp} \times \text{OclEnv} \rightarrow \text{Val} \\
\text{external} & : \text{OclExp} \times \text{OclEnv} \rightarrow \text{OclExp} \\
\text{external} & : \mathcal{L}(\text{OclExp}) \times \text{OclEnv} \rightarrow \mathcal{L}(\text{Val}) \\
\text{external} & : \mathcal{L}(\text{OclExp}) \times \text{OclEnv} \rightarrow \mathcal{L}(\text{OclExp})
\end{align*}
\]

Given an UML model configuration, these functions allow OCL expressions to be evaluated. There are two kinds of evaluation. A complete evaluation \(\llbracket e \rrbracket\) takes an expression and a variable assignment and returns the value that the expression takes in the current configuration. A partial evaluation \(\llbracket e @pre \rrbracket\) takes an expression and a variable assignment and evaluates every expression of the form \("e @pre"\). For both kinds of evaluation, we define point-wise extension to lists (\(\llbracket e \rrbracket_C\) and \(\llbracket e \rrbracket_C\)). This simplifies the evaluation of a list of method parameters, which are OCL expressions.

The evaluation functions require three auxiliary external functions:

\[
\begin{align*}
\text{external} & : \text{bop} : \text{Bop} \times \text{Val} \times \text{Val} \rightarrow \text{Val} \\
\text{external} & : \text{uop} : \text{Uop} \times \text{Val} \rightarrow \text{Val} \\
\text{external} & : \vdash : \text{OclExp} \rightarrow \text{Type}
\end{align*}
\]

The first two give the standard interpretations of binary and unary operators. The \(\vdash\) function assigns a type to an OCL expression. We write \(e \vdash \tau\) when expression \(e\) is of type \(\tau\). This function is defined by derivation rules, as explained in Section C.1.

4.1.2 Semantics

Given the required external functions, the semantics of an OCL expression is defined as follows:

\textbf{External Function 4.1 (Evaluating an Expression)} Consider an OCL expression \(e\) and a variable environment \(\zeta\). The evaluation function \(\llbracket e \rrbracket_\zeta\) is defined as follows. In the case of values, variables and simple operators, we have:
\[
\begin{align*}
e & \equiv v : v \\
e & \equiv x : \zeta(x) \\
e & \equiv e_1 \oplus e_2 : \text{bop}(\Delta, [e_1]_\zeta, [e_2]_\zeta) \\
e & \equiv \Delta e_1 : \text{uop}(\Delta, [e_1]_\zeta)
\end{align*}
\]

In the case of navigation, i.e. \( e \equiv e_1 \cdot cf \), \([e]_\zeta\) is defined according to the three following cases:

\[
\begin{align*}
\text{if } e_1 \vdash C \text{ then} & \quad \text{heap}([e_1]_\zeta).\text{env}(cf) \\
\text{if } e_1 \vdash l(C) \text{ and } cf.f.ty \in \text{SimpleType} \text{ then} & \quad \text{let } (o_1, \ldots, o_n) = [e_1]_\zeta \text{ in} \\
& \quad (\text{heap}(o_1).\text{env}(cf), \ldots, \text{heap}(o_n).\text{env}(cf)) \\
\text{if } e_1 \vdash l(C) \text{ and } cf.f.ty \in \text{ListType} \text{ then} & \quad \text{let } (o_1, \ldots, o_n) = [e_1]_\zeta \text{ in} \\
& \quad \text{heap}(o_1).\text{env}(cf) @ \ldots @ \text{heap}(o_n).\text{env}(cf)
\end{align*}
\]

Note that \( cf.f.ty \) corresponds to the declared type of a field.

If \( e \equiv e_1 \cdot \text{iterate}\{x_1; x_2 = e_2 \mid e_3\} \), then \([e]_\zeta\) is defined as follows:

\[
\begin{align*}
\text{let } v_1 & = [e_1]_\zeta \text{ in} \\
& \quad \text{if } \text{size}(v_1) = 0 \text{ then} \\
& \quad \quad \emptyset \\
& \quad \text{else} \\
& \quad \quad \text{let } v_2 = [e_2]_\zeta \text{ in} \\
& \quad \quad \text{let } v_3 = \text{hd}(v_1) \text{ in} \\
& \quad \quad \text{let } v_4 = [e_3]_\zeta[x_1 \mapsto v_3, x_2 \mapsto v_2] \text{ in} \\
& \quad \quad \text{if } \text{size}(v_1) = 1 \text{ then} \\
& \quad \quad \quad v_4 \\
& \quad \quad \text{else} \\
& \quad \quad \quad \text{[tail}(v_1).\text{iterate}\{x_1; x_2 = v_4 \mid e_3\}]_\zeta
\end{align*}
\]

External Function 4.2 (Partially Evaluating an Expression) Consider an OCL expression \( e \) and a variable environment \( \zeta \). In the case of values and variables, we have \([e]_\zeta = e\). The other cases are defined as follows:
Finally, when \( e \equiv e_1@pre \), the subexpression is evaluated, i.e. \( \llbracket e_1@pre \rrbracket_\zeta = [e_1]_\zeta \).

The point-wise extensions to list are defined straightforwardly as described below.

**External Function 4.3 (Evaluating a List of Expressions)**  Consider a list of OCL expressions \( \langle e_1, \ldots, e_n \rangle \) and a variable environment \( \zeta \). The function \( \llbracket \rrbracket_\zeta^L \) is defined as the point-wise extension of \( \llbracket \rrbracket_\zeta \), that is:

\[
\llbracket \langle e_1, \ldots, e_n \rangle \rrbracket_\zeta^L = \langle \llbracket e_1 \rrbracket_\zeta, \ldots, \llbracket e_n \rrbracket_\zeta \rangle
\]

**External Function 4.4 (Partially Evaluating a List of Expressions)**  Consider a list of OCL expressions \( \langle e_1, \ldots, e_n \rangle \) and a variable environment \( \zeta \). The \( \llbracket \rrbracket_\zeta^L \) is defined as the point-wise extension of \( \llbracket \rrbracket_\zeta \), that is:

\[
\llbracket \langle e_1, \ldots, e_n \rangle \rrbracket_\zeta^L = \langle \llbracket e_1 \rrbracket_\zeta, \ldots, \llbracket e_n \rrbracket_\zeta \rangle
\]

### 4.2 Constraints

#### 4.2.1 Syntax

An OCL constraint is defined by a context (a class and optionally a method signature), a list of OCL expressions acting as atomic properties and a \( \mu \)-formula that expresses the temporal relation between these properties.

The syntax of a \( \mu \)-formula is given by Fig. 4.2. The symbol \( \Phi \) denotes an ASM first-order predicate. The symbol \( X \) is a \( \mu \)-variable. The construct \( X.\phi \) means: from the current configuration, a transition exists leading to a configuration where \( \phi \) holds. The construct \( \nexists X.\phi \) means: from the current configuration, every possible transition leads to a configuration where \( \phi \) holds. The constructs \( \mu X.\phi \) and \( \nu X.\phi \) allows \( \phi \) to be iterated. It is assumed that \( \phi \) contains the free variable \( X \). Thus, \( \phi \) will be tested for successive values of \( X \), which is intuitively considered as a set of current configurations. In particular, \( \mu X.\phi \) is intuitively considered finitely iterating \( \phi \) and \( \nu X.\phi \) is intuitively considered as infinitely iterating \( \phi \).

ASM predicates are used to fetch the value of evaluated OCL expressions and to check the status of method instances. That information is stored in three dynamic functions:

<table>
<thead>
<tr>
<th>dynamic</th>
<th>\texttt{CVals}</th>
<th>( \text{Obj} \rightarrow \mathcal{L}(Val) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>dynamic</td>
<td>\texttt{@now}</td>
<td>( \text{Obj} \times \text{MSig} \rightarrow \text{Bool} )</td>
</tr>
<tr>
<td>dynamic</td>
<td>\texttt{@post}</td>
<td>( \text{Obj} \times \text{MSig} \rightarrow \text{Bool} )</td>
</tr>
</tbody>
</table>
In any configuration, we assume that these functions correctly depict atomic properties of specified constraints and method instances. They are evaluated during the computation of the UML model execution graph as explained in Section 4.3. For definiteness, we restrict ASM predicates.

**Definition 4.2 (Allowed Predicates)** A predicate \( \Phi \) of a \( \mu \)-formula has one of the following form:

- **true**
- **false**
- **\( CVals(o).n \)** where \( o \) is a free variable and \( n \) an integer
- **\( \circledast n(o,m) \equiv \exists m',Msig \cdot (c,m') \preceq_o (c,m) \wedge \circledast now(o,m') \)** where \( o \) is a free variable, \( m \) a method signature and \( c \) a class
- **\( \circledast p(o,m) \equiv \exists m',Msig \cdot (c,m') \preceq_o (c,m) \wedge \circledast post(o,m') \)** where \( o \) is a free variable, \( m \) a method signature and \( c \) a class.

The \( \circledast \) operator applied to \( CVals(o) \) allows us to retrieve the \( n^{th} \) element of the list. Notice how the \( \circledast n(o,m) \) and \( \circledast p(o,m) \) predicates take into consideration overridden methods. This is necessary as a constraint applies to subclasses and overridden methods.

The abstract syntax of an OCL constraint is defined below.

**Definition 4.3 (OCL Constraint)** An OCL constraint is a 4-tuple \((C, m, \text{exp}, \phi)\), where:

- **\( C : Class \)** is the class context of the constraint
- **\( m : Msig \)** is the (optional) method context of the constraint
- **\( \text{exp} \)** is a list of OCL expressions
- **\( \phi \)** is \( \mu \)-formula containing only allowed ASM predicates (using \( C \) as the class and \( m \) as the method signature).

### 4.2.2 Semantics

A UML model satisfies an OCL constraint \((C, m, \text{exp}, \phi)\) if the \( \mu \)-formula is satisfied for every instance of \( C \) (and its subclasses). The free variable in the predicates of the \( \mu \)-formula is a place-holder for an instance of \( C \). The formal definition requires the observation criteria of Section 2.5. In addition, it requires the relation \( \Lambda \models \phi \), which means that the graph \( \Lambda \) satisfies the formula \( \phi \). This corresponds to the formal semantics of the adapted \( \mu \)-calculus, which is defined in Section C.2.
Definition 4.4 (OCL Constraints Semantics) Consider an UML execution graph $\Lambda$ and an OCL constraint $c = (C, MSig, \text{exps}, \phi)$ with $r$ as the only free variable of the predicates of $\phi$. The graph $\Lambda$ satisfies $c$ if:

- For every $o \in \text{Obj}$ such that $\exists \overrightarrow{x} \in \Lambda. \text{heap}(o)^{\overrightarrow{x}} \neq \bot \land \text{heap}(o)^{\overrightarrow{x}}.c \preceq_h C$, the following holds:

  $\Lambda|_{\text{heap}(o)\neq\bot} \models \phi[r \mapsto o].$

The expression “$[r \mapsto o]$” denotes the substitution of the free variable $r$ with the object $o$ in the predicates of $\phi$. Note how the graph is observed to make sure the formula is verified on the segment of the graph where the object exists.

4.2.3 Templates

Class Invariants A class invariant template is translated to an OCL constraint as follows:

$$\text{context } C \quad \text{inv: } e_1 \quad \mapsto \quad (C, \bot, \{e_1\}, \nu X. \Box X \land CVals(o).1)$$

Informally, it reads as: the OCL expression $e_1$ always holds for an object of type $C$.

Method Pre/post Conditions A pre/post condition template is translated to an OCL constraint as follows:

$$\text{context } C::\text{MSig} \quad \text{pre: } e_1 \quad \text{post: } e_2 \quad \mapsto \quad \phi \equiv \circ \ominus_n(o, m) \Rightarrow CVals(o).1 \quad \psi \equiv \ominus_p(o, m) \Rightarrow CVals(o).2$$

Informally, it reads as: if the system is in a $\ominus$pre configuration, then $e_1$ holds; if the system is in a $\ominus$post configuration, then $e_2$ holds. Note how the $\ominus$pre configuration is encoded as: there exists a next configuration where $\ominus$now holds.

Specifying Other Constraints According to the semantics developed, OCL constraints are not restricted to invariants and pre/post conditions. For example, an after/ eventually template may be translated to an OCL constraint as follows:
Informally, it reads as: if \( e_1 \) holds in some configurations, then \( e_2 \) will eventually hold in a future configuration. The \( \mu \)-formula is more intricate. It first reads as: it is always true that \( \phi \Rightarrow \psi \). Then \( \phi \) is \texttt{true} if \( e_1 \) holds. The formula \( \psi \) may be understood as follows. For a particular path, if a future configuration is found such that \( e_2 \) holds, then \( \psi \) is \texttt{true} for that path. If a future configuration has not yet been found on the path such that \( e_2 \) holds and there is no more successor configuration (i.e. the use of \texttt{true}), then \( \psi \) is \texttt{false} on that path. The formula \( \psi \) must hold for every path.

### 4.3 UML Model Semantics Extensions

The OCL \texttt{@pre} operator requires that the \texttt{@pre} configuration be matched with the corresponding \texttt{@post} configuration. This is not easily expressed by a \( \mu \)-formula. In fact, it may even be impossible but it is beyond the scope of this document to prove such a claim. To overcome the problem, we evaluate the OCL expressions of an OCL constraint during the computation of the UML model execution graph. We push the partially evaluated OCL expressions on the stack. This insures the required correspondence.

This can be done with simple extensions to the semantics. First, we extend the \( \Upsilon_{\text{stat}} \) vocabulary with a function containing a list of OCL expressions associated with a class:

\[
\text{static} \quad \text{CONS} : \text{Class} \rightarrow \mathcal{L}(\text{OclExp})
\]

Then we extend the \( \Upsilon_{\text{uml}} \) vocabulary with a function containing a list of evaluated OCL expressions:

\[
\text{dynamic} \quad \text{CExp} : \text{Obj} \rightarrow \mathcal{L}(\text{OclExp})
\]

Finally, we redefine the sort \( \text{Frame} \) to include a list of partially evaluated OCL expressions:

\[
\text{sort} \quad \text{Frame} = \mathcal{T}(\text{mth} : \text{ClassMSig}, \ldots, \text{ocl} : \mathcal{L}(\text{OclExp}))
\]

**Method Instances** The functions \texttt{@now} and \texttt{@post} need to be initialized at the beginning of the step by the rule \( \Pi_{\text{chooseThread}} \):

\[
\| \Sigma_w:\text{Obj} : \Sigma_{m:\text{Msig}} : \text{@now}(o, m) := \text{false} \\
\| \Sigma_w:\text{Obj} : \Sigma_{m:\text{Msig}} : \text{@post}(o, m) := \text{false}
\]

Then, when a method is pushed on the stack, the \texttt{@now} function is updated. The partially evaluated OCL expressions are also pushed on the stack. This is done in the rule \( \Pi_{\text{pushMethod}(c, msig, \ldots)} \):
When a method is popped, the rule $\Pi_{\text{popMethod}}$ updates the $\texttt{@post}$ function. Also, the partially evaluated OCL expressions are taken from the frame and put into the $C\text{Exps}$ function:

\[
\| [...]
\| \texttt{tf} := \texttt{top}(cont(\tilde{h})) ;
\| \texttt{CExs}(\tilde{\alpha}) := \texttt{tf}.ocl
\| \texttt{@post}(\tilde{\alpha}, \texttt{tf}.cmsig,msig) := \texttt{true}
\]

**Object Creation** The object creation rule $\Pi_{\text{createObj}}(o, cf, C)$ is extended by adding a list of OCL expressions for the newly created object:

\[
\partial o' : \textit{Obj} ;
\| [...]
\| \texttt{CExs}(o') := \texttt{cons}(C)
\]

**Evaluating Atomic Properties** Finally, the evaluation of atomic properties is done by the $\Pi_{\text{fireUpdates}}$ rule:

\[
\| [...]
\| \Sigma_\alpha:Obj : \textit{CVals}(o) := \llbracket CExs(o) \rrbracket_c^\zeta
\]

$\llbracket CExs(o) \rrbracket_c^\zeta$
Bibliography


Appendix A

ASM Details

This appendix contains details about the ASM formalism. We define the notion of domain. We give the formal semantics of ASM predicates and of transition rules.

A.1 Term Evaluation

Variable Assignment Evaluating a term requires a state and a variable assignment. A variable assignment is a finite mapping from variables of an ASM transition rule and elements of the appropriate carrier sets of a current state. The initial mapping $\zeta$ maps every variable to $\bot^\mathfrak{M}$. Updating the variable assignment is as usual.

Definition 1.1 The update of $\zeta^*$ on $x$ is defined as follows:

$$\zeta^*_{x\rightarrow a}(v) = \begin{cases} a & \text{if } v = x \\ \zeta^*(v) & \text{otherwise} \end{cases}$$

Term Interpretation The terms are evaluated recursively. We define a function $[\cdot]^\mathfrak{M}_\zeta : \Delta \rightarrow \bigcup_{i \in I} S_i^\mathfrak{M}$ to denote this evaluation.

Definition 1.2 Consider a state $\mathfrak{M}$ and a variable assignment $\zeta^*$, the term evaluation function $([\cdot]^\mathfrak{M}_\zeta)$ is defined as follows:

- $[v_i]^\mathfrak{M}_\zeta = \zeta^*(v_i)$
- $[f]^\mathfrak{M}_\zeta = f^\mathfrak{M}$
- $[[f(t_1, \ldots, t_k)]^\mathfrak{M}_\zeta = f^\mathfrak{M}([t_1]^\mathfrak{M}_\zeta, \ldots, [t_k]^\mathfrak{M}_\zeta)$. 

47
A.2 Domains

In general, a carrier set is infinite. We define the notion of domain, which is a finite subset of that carrier set. Formally, this is as follows:

**Definition 1.3 (Domain)** Let \( \mathcal{A} \) be a state and \( S_i \) be a sort. The domain \( \tilde{S}^\mathcal{A}_{S_i} \) of \( S_i \) in \( \mathcal{A} \) is defined as follows:

\[
\tilde{S}^\mathcal{A}_{S_i} = \{ a \in S_i^\mathcal{A} \mid \exists f \in \Omega_{stat} \cup \Omega_{dyn} \cdot f^\mathcal{A}(\ldots, a, \ldots) \neq \perp^\mathcal{A} \lor f^\mathcal{A}(\ldots) = a \}
\]

A.3 Predicate Semantics

Given a state \( \mathcal{A} \) and a variable assignment \( \zeta^* \), Fig. A.1 gives the semantics of a predicate \( \Phi \). The evaluation function \( \llbracket \cdot \rrbracket^\mathcal{A}_\zeta \) takes a predicate and returns a boolean value (i.e. an element of sort \( \text{Bool} \)). The only subtlety is that quantifiers are bounded to domains rather than ranging over the whole carrier set.

\[
\begin{align*}
\llbracket v \rrbracket^\mathcal{A}_\zeta & \iff \zeta^*(v) \\
\llbracket f(t) \rrbracket^\mathcal{A}_\zeta & \iff \llbracket f(t) \rrbracket^\mathcal{A}_{\zeta} \\
\llbracket -\Phi \rrbracket^\mathcal{A}_\zeta & \iff \neg \llbracket \Phi \rrbracket^\mathcal{A}_{\zeta} \\
\llbracket \Phi \land \Psi \rrbracket^\mathcal{A}_\zeta & \iff \llbracket \Phi \rrbracket^\mathcal{A}_{\zeta} \land \llbracket \Psi \rrbracket^\mathcal{A}_{\zeta} \\
\llbracket \exists x : S_i. \Phi \rrbracket^\mathcal{A}_\zeta & \iff \llbracket \Phi \rrbracket^\mathcal{A}_{\zeta[a \mapsto a]} \text{ for some } a \in \tilde{S}^\mathcal{A}_{S_i} \\
\llbracket \forall x : S_i. \Phi \rrbracket^\mathcal{A}_\zeta & \iff \llbracket \Phi \rrbracket^\mathcal{A}_{\zeta[a \mapsto a]} \text{ for all } a \in \tilde{S}^\mathcal{A}_{S_i} \\
\llbracket \Phi \lor \Psi \rrbracket^\mathcal{A}_\zeta & \iff \llbracket \neg(\neg\Phi \land \neg\Psi) \rrbracket^\mathcal{A}_\zeta \\
\llbracket \Phi \Rightarrow \Psi \rrbracket^\mathcal{A}_\zeta & \iff \llbracket \neg\Phi \lor \Psi \rrbracket^\mathcal{A}_\zeta \\
\llbracket \Phi \Leftrightarrow \Psi \rrbracket^\mathcal{A}_\zeta & \iff \llbracket \Psi \Rightarrow \Phi \rrbracket^\mathcal{A}_\zeta \\
\end{align*}
\]

Figure A.1: Predicate Semantics

A.4 Rule Semantics

The semantics of transition rules is given in terms of the function \( \llbracket \cdot \rrbracket^\mathcal{A}_\zeta \) which, given a state \( \mathcal{A} \) and the initial variable assignment \( \zeta \), takes a rule and returns a collection of update sets. The calculus of Fig. A.2 defines the semantics function Note that in the \( \Sigma \) and \( \epsilon \) rules, we assume that sets in \( (I_a)_{a \in J} \) are pairwise different, i.e. \( \forall a, a' \in J \cdot I_a \cap I_a' = \emptyset \). The function yields a collection of update sets of form \( (U_i)_{i \in J} \). The reason is that many transitions may occur from one state as an effect of the non-deterministic choose rule \( (\epsilon_{x,S}, \Phi : \Pi) \).
\[ [\cdot]_{\mathcal{A}}^\mathcal{M} \rightarrow \{\emptyset\} \]

\[ [f(\tilde{a}) := s]_{\mathcal{A}}^\mathcal{M} \rightarrow \{((f, \tilde{a}), b)\} \quad \text{if } \tilde{a} = [f]_{\mathcal{A}}^\mathcal{M} \text{ and } b = [s]_{\mathcal{A}}^\mathcal{M} \]

\[ [\Pi_1]_{\mathcal{A}}^\mathcal{M} \rightarrow (U_i)_{i \in I} \]

\[ \Phi \Pi_1 \Pi_2]_{\mathcal{A}}^\mathcal{M} \rightarrow (V_j)_{j \in J} \quad \text{if } \phi \Phi \Pi_1 \Pi_2]_{\mathcal{A}}^\mathcal{M} = \text{true} \]

\[ [\Pi_1 \Pi_2]_{\mathcal{A}}^\mathcal{M} \rightarrow (U_i)_{i \in I} \quad \text{if } \phi \Phi \Pi_1 \Pi_2]_{\mathcal{A}}^\mathcal{M} = \text{false} \]

\[ [\Pi_1 || \Pi_2]_{\mathcal{A}}^\mathcal{M} \rightarrow (U_i)_{i \in I} \quad \text{for each } a \in J \]

\[ \Sigma \epsilon \; [\Pi]_{\mathcal{A}}^\mathcal{M} \rightarrow \bigcup_{i \in I} U_i \quad \text{if } J = \{a \in S_i \mid [\phi]_{\mathcal{A} = a}^\mathcal{M} = \text{true}\} \]

\[ [\Pi]_{\mathcal{A}}^\mathcal{M} \rightarrow (U_i)_{i \in I} \quad \text{for each } a \in J \]

\[ \epsilon \Sigma \; [\Pi]_{\mathcal{A}}^\mathcal{M} \rightarrow \bigcup_{i \in I} U_i \quad \text{if } J = \{a \in S_i \mid [\phi]_{\mathcal{A} = a}^\mathcal{M} = \text{true}\} \]

\[ [\Pi]_{\mathcal{A}}^\mathcal{M} \rightarrow (U_i)_{i \in I} \quad \text{if } a = [t]_{\mathcal{A}}^\mathcal{M} \]

\[ [x := t]_{\mathcal{A}}^\mathcal{M} \rightarrow (U_i)_{i \in I} \quad \text{if } a = [t]_{\mathcal{A}}^\mathcal{M} \]

\[ \text{A.5 External Functions} \]

Given an \texttt{ASM} \( M \) containing the sort \( S_i \), then we assume that \( S_i \) is totally ordered by a relation \( \preceq_{S_i} \) and that \( \Omega \) contains the following external functions:

\begin{align*}
\text{external} & \quad \cdot = \cdot : S_i \times S_i \rightarrow \text{Bool} \\
\text{external} & \quad \cdot \preceq_{S_i} \cdot : S_i \times S_i \rightarrow \text{Bool}
\end{align*}

The following external functions are assumed to be correctly defined on a constructed sort \( S(A) \):

\begin{align*}
\text{external} & \quad \emptyset : S(A) \\
\text{external} & \quad \cup : S(A) \times S(A) \rightarrow S(A) \\
\text{external} & \quad \cap : S(A) \times S(A) \rightarrow S(A) \\
\text{external} & \quad \in : A \times S(A) \rightarrow \text{Bool} \\
\text{external} & \quad \{\cdot\} : A \rightarrow S(A)
\end{align*}
The following external functions are assumed to be correctly defined on a constructed sort $\mathcal{L}(A)$:

\begin{align*}
\text{external} & \quad \emptyset : \mathcal{L}(A) \\
\text{external} & \quad \cdot \circ \cdot : \mathcal{L}(A) \times \mathcal{L}(A) \rightarrow \mathcal{L}(A) \\
\text{external} & \quad \cdot : \cdot : A \times \mathcal{L}(A) \rightarrow \mathcal{L}(A) \\
\text{external} & \quad \cdot \in \cdot : A \times \mathcal{L}(A) \rightarrow \text{Bool} \\
\text{external} & \quad \text{hd} : \mathcal{L}(A) \rightarrow A \\
\text{external} & \quad \text{tail} : \mathcal{L}(A) \rightarrow \mathcal{L}(A) \\
\text{external} & \quad \langle \cdot \rangle : A \rightarrow \mathcal{L}(A)
\end{align*}

The following external functions are assumed to be correctly defined on a constructed sort $T(A_1, \ldots, A_n)$:

\begin{align*}
\text{external} & \quad \pi_1 : T(A_1, \ldots, A_n) \rightarrow A_1 \\
\text{external} & \quad \cdot : \cdot \\
\text{external} & \quad \pi_n : T(A_1, \ldots, A_n) \rightarrow A_n
\end{align*}

The functions $\pi_i$ correspond to the projection operators on tuples.

The following external functions are assumed to be correctly defined on a constructed sort $A_1 \rightarrow A_2$:

\begin{align*}
\text{external} & \quad \emptyset : A_1 \rightarrow A_2 \\
\text{external} & \quad \cdot \in \cdot : A_1 \times (A_1 \rightarrow A_2) \rightarrow \text{Bool} \\
\text{external} & \quad \cdot(\cdot) : (A_1 \rightarrow A_2) \times A_1 \rightarrow A_2 \\
\text{external} & \quad \{ \cdot \rightarrow \cdot \} : A_1 \times A_2 \rightarrow (A_1 \rightarrow A_2) \\
\text{external} & \quad \cdot \oplus \cdot : (A_1 \rightarrow A_2) \times (A_1 \rightarrow A_2) \rightarrow (A_1 \rightarrow A_2)
\end{align*}

The function $\cdot(\cdot)$ searches a finite mapping $A_1 \rightarrow A_2$ for a particular value $a : A_1$. If there exists a couple $(a, b) : (A_1, A_2)$ in the mapping, then $b$ is returned. Otherwise, $\bot$ is returned.

The second function $\oplus$ concatenates two mappings. It is defined as follows:

\textbf{External Function 1.1 (Mapping Concatenation)} \quad \text{Given two mappings } m_1 : A_1 \rightarrow A_2 \text{ and } m_2 : A_1 \rightarrow A_2, \text{ the concatenation is defined as follows:}

\begin{align*}
m_1 \oplus m_2(a) &= \begin{cases} 
m_2(a) & \text{if } m_2(a) \neq \bot \\
m_1(a) & \text{else}
\end{cases}
\end{align*}

The following external functions are assumed to be correctly defined on a constructed sort $A^*$:

\begin{align*}
\text{external} & \quad \emptyset : A^* \\
\text{external} & \quad \cdot \in \cdot : A \times A^* \rightarrow \text{Bool} \\
\text{external} & \quad \text{push} : A^* \times A \rightarrow A^* \\
\text{external} & \quad \text{pop} : A^* \rightarrow A^* \\
\text{external} & \quad \text{top} : A^* \rightarrow A
\end{align*}

These functions correspond to the usual operators on stacks, except for $\cdot(\cdot \rightarrow \cdot)$.

Finally, we define an external function taking two lists and returning a mapping.


**APPENDIX A. ASM DETAILS**

external \texttt{buildMap} : \mathcal{L}(A) \times \mathcal{L}(B) \rightarrow A \mapsto B

**External Function 1.2 (Building a Mapping)**

Given two lists \( l_1 = \langle a_1, \ldots, a_n \rangle \) and \( l_2 = \langle b_1, \ldots, b_n \rangle \):

\[
\text{buildMap}(l_1, l_2) = \{ a_1 \mapsto b_1, \ldots, a_n \mapsto b_n \}
\]

A.6 Extend Rule

The extend rule “\( \partial_{x;S_i} : \Pi \)" binds \( x \) to an element \( s \in S^\mathfrak{f}_i \setminus S^\mathfrak{g}_i \) and evaluates \( \Pi \). It is a shortcut for \( x = \partial_{S_i} : \Pi \) , where \( \partial_{S_i} \) is an external function that returns the biggest element (according to \( \leq_{S_i} \)) not in the domain of \( S_i \). Removing an element from the domain of \( S_i \) is achieved by correctly updating dynamic functions to the undefined element.
Appendix B

UML Semantics Details

This chapter presents the complete specification of the UML model semantics developed in this document. It is presented as shortly as possible and comments are added only when deemed necessary. The $\mathcal{Y}_{\text{stat}}$, $\mathcal{Y}_{\text{uml}}$ and $\mathcal{Y}_{\text{aux}}$ vocabularies are presented along with the corresponding elaborated functions and well-formedness constraints. Lastly, the complete transition rule is presented.

B.1 Static Vocabulary

B.1.1 Extracted Information

The first part of the $\mathcal{Y}_{\text{stat}}$ vocabulary is a straightforward representation of a model’s class and statechart diagrams. Sorts are used to structure how the information is stored while static functions actually hold the diagrams.

Basics

<table>
<thead>
<tr>
<th>sort</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>sort</td>
<td>Obj</td>
</tr>
<tr>
<td>sort</td>
<td>OclExp</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>sort</th>
<th>SimpleType</th>
<th>$\equiv$</th>
<th>${\text{VOID, INT, BOOL}} \cup \text{Class}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>sort</td>
<td>ListType</td>
<td>$\equiv$</td>
<td>${\tau : \text{SimpleType} \mid \text{L}(\tau)} \cup {C : \text{Class} \mid \text{L}(C)}$</td>
</tr>
<tr>
<td>sort</td>
<td>Type</td>
<td>$\equiv$</td>
<td>$\text{SimpleType} \mid \text{ListType}$</td>
</tr>
</tbody>
</table>

Fields
sort FieldName = T(low : Int, up : Int)
sort SimpleField = T(t : SimpleType, f : FieldName)
sort ListField = T(t : ListType, f : FieldName, m : Mult)
sort Field = SimpleField \ ListField
sort ClassField = T(c : Class, f : Field)

Method Signatures

sort Meth
sort ParamName
sort Param = T(t : Type, p : ParamName)
sort MSig = T(mth : Meth, p : L(Param), r : Param)
sort ClassMSig = T(c : Class, msig : MSig)

Class Diagram In the following, note that a class association is mapped to a field named after the supplier end of the association. Also, the direct inheritance relation \(<_d\) and the SUPER function are specified by model’s generalization associations. In the model, a thread is identified by a class stereotype.

static CLASSES : S(Class)
static \(<_d\) : R(Class)
static SUPER : Class \rightarrow Class
static FIELDS : Class \rightarrow S(Field)
static METHODS : Class \rightarrow S(MSig)
static THREADS : S(Class)

Triggers

sort Signal
sort TType = {EMPTY, SIGNAL, TCALL, TRETURN}
sort TEmpty = T(ty : TType)
sort TSignal = T(s : Signal, ty : TType)
sort TCall = T(msig : MSig, ty : TType)
sort TReturn = T(msig : MSig, ty : TType)

Actions

sort AType = {SIGNAL, NEW, DEL, ASSIGN, MRETURN, MCALL}
sort CallType = {VIRTUAL, SUPER}
sort ASignal = T(s : Signal, ty : AType)
sort ANew = T(cf : ClassField, ty : AType)
sort ADel = T(cf : ClassField, ty : AType)
sort AAssign = T(cf : ClassField, exp : OclExp)
sort AReturn = T(msig : MSig, exp : OclExp)
sort ACall = T(rcv : OclExp, cmsig : ClassMSig, exps : L(OclExp),
                cty : CallType, ty : AType)
States

\[
\begin{align*}
\text{sort } StateType & \equiv \{ \text{INIT, FINAL, SIMPLE, OR, AND} \} \\
\text{sort } StateName & \\
\text{sort } State & = T(n : StateName, ty : StateType)
\end{align*}
\]

Transitions

\[
\begin{align*}
\text{sort } TransName & \\
\text{sort } Trigger & = TSignal \mid TCall \mid TReturn \\
\text{sort } Action & = ASignal \mid ANew \mid ADel \mid AAAssign \mid ACall \mid AReturn \\
\text{sort } Guard & = OclExp \\
\text{sort } Trans & = T(n : TransName, src : State, tr : Trigger, g : Guard, \\
& \quad acts : \mathcal{L}(Action), tgt : State)
\end{align*}
\]

Statechart Diagram In the following, the functions TOP, CHILD and UP hold the hierarchical structure of a model’s statechart diagrams. Exceptionally, the function TRANS requires some elaboration. This is explained in Section B.1.4.

\[
\begin{align*}
\text{static } STATES & : \text{Class} \rightarrow S(State) \\
\text{static } TRANS & : S(Trans) \\
\text{static } TOP & : \text{Class} \rightarrow State \\
\text{static } CHILD & : \text{State} \rightarrow S(State) \\
\text{static } UP & : \text{State} \rightarrow State
\end{align*}
\]

B.1.2 Elaborated Information

The second part of the \(\Upsilon_{\text{stat}}\) vocabulary contains elaborated information. Some elaborated function are needed in the dynamic semantics. Others are auxiliary functions used in the definition of the latter.

Class Diagram Elaborated Functions

The dynamic semantics uses the following elaborated functions:

\[
\begin{align*}
\text{static } \preceq_h & : \mathcal{R}(\text{Class}) \\
\text{static } \text{FIELDS}* & : \text{Class} \rightarrow S(ClassField) \\
\text{static } \text{METHODS}* & : \text{Class} \rightarrow S(MSig) \\
\text{static } \text{LOOKUP} & : \text{Class} \times ClassMSig \rightarrow \text{Class}
\end{align*}
\]

The inheritance relation \(\preceq_h\) is the transitive reflexive closure of the direct inheritance relation \(\prec_d\). The function FIELDS* returns every field of a class, including inherited fields. The function METHODS* returns every method of a class, including inherited methods. The following auxiliary functions are necessary for defining the LOOKUP function and well-formedness constraints.
static \subseteq : \mathcal{R}(Type) \\
static \leq_s : \mathcal{R}(MSig) \\
static \mathsf{APP} : \text{ClassMSig} \rightarrow \mathcal{S}(	ext{ClassMSig}) \\
static \mathsf{APP} : \text{ClassMSig} \rightarrow \text{ClassMSig} \\
static \leq_o : \mathcal{R}(\text{ClassMSig}) \\
static \leq^* : \mathcal{R}(\text{ClassMSig})

The fields* and methods* functions are defined straightforwardly using the inheritance relation.

**Elaborated Function 2.1 (Fields of a Class)** Given a class $A$, $\text{FIELDS}^*(A)$ is defined as follows:

$$
\text{FIELDS}^*(A) = \{(B, f) \mid A \leq B \land f \in \text{FIELDS}(B)\}
$$

**Elaborated Function 2.2 (Methods of a Class)** Consider a class $A$. The function $\text{METHODS}^*(A)$ is defined as follows:

$$
\text{METHODS}^*(A) = \{(B, m) \mid A \leq B \land m \in \text{METHODS}(B)\}
$$

Definition of the important $\text{LOOKUP}$ requires more work. First, we need a subtype relation ($\subseteq$). It is defined as follows.

**Elaborated Function 2.3 (Subtype Relation)** Given two types $\tau, \tau' : Type$, the subtype relation is defined as follows:

$$
\tau \subseteq \tau' \iff \begin{cases} 
\tau = \tau' \\
\tau = C, \tau' = D \text{ and } C \leq_h D \\
\tau = \text{l}(C), \tau' = \text{l}(D) \text{ and } C \subseteq D
\end{cases}
$$

In order to define the $\leq_s$ (more specific) relation method signatures are required.

**Elaborated Function 2.4 (More Specific)** Consider two method signatures $cm$ and $cn$, with

$$
\text{cm.msig.p} = \langle (\tau_1^{cm}, p_1^{cm}), (\tau_2^{cm}, p_2^{cm}), \ldots, (\tau_i^{cm}, p_i^{cm}) \rangle
$$

and

$$
\text{cn.msig.p} = \langle (\tau_1^{cn}, p_1^{cn}), (\tau_2^{cn}, p_2^{cn}), \ldots, (\tau_j^{cn}, p_j^{cn}) \rangle
$$

The relation $\leq_s$ is defined as follows.
\[ cm \preceq_s cn \iff cm.msigt.cm = cn.msigt.cm \]
\[ \quad \text{and} \quad cm.c \preceq_h cn.c \]
\[ \quad \text{and} \quad i = j \]
\[ \quad \text{and} \quad \tau_k^{cm} \subseteq \tau_k^{cm} \quad \text{for} \quad k = 1, \ldots, i \]

The more specific relation is also required to define the overriding relation.

**Elaborated Function 2.5** A method \( cm \) directly overrides \( cn \) if there is some class \( B \) such that:

1. \( cm.msigt.cm = cn.msigt.cm \)
2. \( cm.c \prec_h B \)
3. \( cn \in \text{METHODS}^+ (B) \)

The relation \( \preceq^*_o \) is the transitive reflexive closure of the direct overriding relation. Finally, we can define the method lookup function.

**Elaborated Function 2.6**

\[
\text{LOOKUP}(A, cm) =
\begin{cases} 
\ A & \text{if there is some } n \in \text{METHODS}(A) \text{ such that } n \preceq^*_o cm.msigt.cm \\
\ C & \text{if } A \prec_d B \text{ and } C = \text{LOOKUP}(B, cmsigt) \\
\bot & \text{otherwise}
\end{cases}
\]

In addition, we define a set of applicable methods in terms of a desired method call. This is used in the elaboration of the method call actions as described in Section B.1.4.

**Elaborated Function 2.7** Given the desired method call \( cm \), the set of applicable methods is defined as follows:

\[
\text{APP}(cm) = \{ cn \mid cm \preceq_s cn \land cn \in \text{METHODS}^+ (cm.c) \}
\]

This allows us to define the most applicable method for a given call, if it exists.

**Elaborated Function 2.8 (Most Applicable Method)** The applicability predicate is defined as follows:

\[
\text{APP}(cm) = cn \text{ such that } \forall cp \in \text{APP}(cm), cn \preceq_s cp
\]
Statechart Diagram Elaborated Functions

The following elaborated function are used in the dynamic semantics:

- **static DEACTIVATES**: \( \text{Trans} \rightarrow \mathcal{S}(\text{State}) \)
- **static ACTIVATES**: \( \text{Trans} \rightarrow \mathcal{S}(\text{State}) \)
- **static INITSTATES**: \( \text{Class} \rightarrow \mathcal{S}(\text{State}) \)
- **static \( \odot \)**: \( \mathcal{R}(\text{Trans}) \)

The \( \odot \) relation indicates that two transitions are in conflict with respect to the state they deactivate. Definition of these functions requires the following auxiliary functions:

- **static CHILD\(^*\)**: \( \text{State} \rightarrow \mathcal{S}(\text{State}) \)
- **static ENC**: \( \text{Trans} \rightarrow \mathcal{S}(\text{State}) \)
- **static SCOPE**: \( \text{Trans} \rightarrow \text{State} \)
- **static ENTERS**: \( \text{State} \times \text{Trans} \rightarrow \mathcal{S}(\text{State}) \)

The function CHILD\(^*\) returns every child of a state. It is defined inductively as follows.

**Elaborated Function 2.9 (State Hierarchy)**

\[
\text{CHILD}\(^*\)(s) = \left\{ \begin{array}{ll} \{s\} & \text{if } \text{CHILD}(s) = \{s\} \\ \bigcup s' \in \text{CHILD}(s) \cdot \text{CHILD}\(^*\)(s') & \text{else} \end{array} \right.
\]

The function ENC returns every state that contains both the source state and the target state of a transition.

**Elaborated Function 2.10 (States Enclosing a Transition)**

\[
\text{ENC}(t) = \{s : \text{State} \mid t.\text{src} \in \text{CHILD}\(^*\)(s) \land t.\text{tgt} \in \text{CHILD}\(^*\)(s)\}
\]

The scope of a transition is its lowest enclosing state, which is unique if the statechart is well-formed.

**Elaborated Function 2.11 (Scope of a Transition)**

\[
\text{SCOPE}(t) = s \text{ such that } s \in \text{ENC}(t) \text{ and } \forall s' \in \text{ENC}(t) . s \in \text{CHILD}\(^*\)(s')
\]

The function ENTERS returns the states that are activated upon entering a state \( s_1 \) as a result of the firing of a transition with target state \( s_2 \). As a shorthand notation, we use \( \text{CHILD}(s) \) instead of \( \text{CHILD}(s) \setminus \{s\} \).

**Elaborated Function 2.12 (Entering a State)**

Given a transition \( t \) and a state \( s \), ENTERS is inductively defined as follows:

\[
\text{ENTERS}(s_1, s_2) =
\]
The main elaborated functions are defined below.

**Elaborated Function 2.13 (Deactivated States)** Given a transition \( t \), the function \textsc{deactivates} is defined as follows:

\[
\text{deactivates}(t) = \begin{cases} 
\text{CHILD}^*(\text{scope}(t)) & \text{if } t.tgt.ty \neq \text{FINAL} \\
\text{CHILD}^*(\text{scope}(t)) & \text{if } t.ttgt.ty = \text{FINAL} \\
& \text{and } \uparrow(\uparrow(t.tgt)).ty \neq \text{AND} \\
\text{CHILD}^*(\text{scope}(t)) & \text{if } t.ttgt.ty = \text{FINAL} \\
& \text{and } \uparrow(\uparrow(t.ttgt)).ty = \text{AND} \\
& \text{and } \uparrow(\uparrow(t.ttgt)) \in \text{scope}(t) \\
\text{CHILD}^*(\uparrow(\uparrow(t.ttgt))) & \text{else}
\end{cases}
\]

**Elaborated Function 2.14 (Activated States)** Given a transition \( t \), \textsc{activates} is defined as follows:

\[
\text{activates}(t) = \begin{cases} 
\text{ENTERS}^{\text{scope}(t)}, t.ttgt & \text{if } t.ttgt.ty \neq \text{FINAL} \\
\emptyset & \text{else}
\end{cases}
\]

**Elaborated Function 2.15 (Initial States)** Given a class \( C \), \textsc{initstates} is defined as follows:

\[
\text{initstates}(C) = \bigcup_{C \subseteq_n D} \text{initstates}(\text{top}(D))
\]

**Elaborated Function 2.16 (Conflicting Transitions)** Two transitions \( t_1, t_2 \) are in conflict if:

\[
t_1 \otimes t_2 \iff t_1 \neq t_2 \text{ and } \text{deactivates}(t_1) \cap \text{deactivates}(t_2) \neq \emptyset
\]

**B.1.3 Well-Formedness**

The following well-formedness constraints apply to the \( \mathcal{Y}_{\text{stat}} \) vocabulary. The notation \( \text{expr} \vdash \tau \) means that the OCL expression \( \text{expr} \) is assigned type \( \tau \) by the typing rules of Section C.1.
**OCL Expression Well-formed Model 2.1**  
An OCL expression is well-formed if it is well-typed.

### Class Diagram Well-Formedness

**Well-formed Model 2.2 (Direct Inheritance Relation)**  
A class has only one superclass:

\[
\text{if } A \prec_d B \text{ and } A \prec_d C \text{ then } B = C
\]

**Well-formed Model 2.3 (Direct Inheritance Relation)**  
The inheritance \( \preceq_h \) relation must be acyclic:

\[
\text{if } A \preceq_h B \text{ and } B \preceq_h A \text{ then } A = B
\]

### Action Well-Formedness

**Well-formed Model 2.4 (Well-formed Action)**  
Consider a transition \( t \) and an action \( a \in t.a \). Assume that \( \text{scope}(t) \in \text{states}(C) \). The action is well-formed if:

- \( a.ty = \text{SIGNAL} \)
- \( a.ty = \text{NEW} \text{ and } a.cf \in \text{FIELDS}^*(C) \)
- \( a.ty = \text{DEL} \text{ and } a.cf \in \text{FIELDS}^*(C) \)
- \( a.ty = \text{ASSIGN}, a.cf \in \text{FIELDS}^*(C), \)
  - \( a.exp \vdash \tau : \text{Type} \text{ and } \tau \subseteq a.cf.ty \)
- \( a.ty = \text{MCALL}, \)
  - \( a.rcv \vdash D \)
  - \( a.exps = \langle e_1, \ldots, e_n \rangle \)
  - \( e_i \vdash \tau_i \text{ for } i = 1 \ldots n \text{ and } \)
  - \( \text{APP}(D, (a.cmsig.msg, \langle \tau_1, \ldots, \tau_i, \ldots, \tau_n \rangle)) \neq \bot \)
- \( a.ty = \text{MRETURN}, \)
  - \( a.msg \in \text{METHODS}^*(C), \text{ and } \)
  - \( a.exp \vdash \tau \text{ and } \tau \subseteq a.msg.r.ty \)
Transition Well-Formedness

Well-formed Model 2.5 (Well-formed Guard)  Consider a transition $t$ and the guard $t.g$. Assume that $\text{scope}(t) \in \text{states}(C)$. The guard is well-formed if:

$$t.g \vdash \text{BOOL}$$

i.e. if it has the type BOOL.

State Diagram Well-Formedness

Well-formed Model 2.6 (OR-state Structure)  An OR-state $s$ is well-formed if:

$$\exists s' \in \text{child}(s) . s'.ty = \text{INIT}$$

Well-formed Model 2.7 (AND-state Structure)  An AND-state $s$ is well-formed if:

$$\text{child}(s) \neq \emptyset \text{ and } \forall s' \in \text{child}(s) . s'.ty = \text{OR}$$

Well-formed Model 2.8 (State Type)

$$\forall s : \text{State} . \text{child}(s) \neq \emptyset \Rightarrow s.ty = \text{OR} \text{ or } s.ty = \text{AND}$$

Well-formed Model 2.9 (State Hierarchy)

$$\forall s, s' : \text{State} . \ s \neq s' \Rightarrow \text{child}(s) \cap \text{child}(s') = \emptyset$$
and
$$\text{child}(s) = \text{child}(s') \Rightarrow s = s'$$
and
$$\text{up}(s) = s' \Rightarrow s \in \text{child}(s')$$
and
$$\text{up}(s) = \bot \iff \exists C : \text{Class} . s = \text{top}(C)$$

Well-formed Model 2.10 (Transitions & Statecharts)

$$\forall t : \text{trans} . \exists C \in \text{classes} . \text{scope}(t) \in \text{states}(C)$$
Well-Formed Statechart  Defining whether a transition is well-formed with respect to the states it activates and deactivates requires the notion of state configuration. A state configuration $SC : S(State)$ is a set of active states for a given statechart.

Well-formed Model 2.11 (AND-state Configuration)  A state configuration $SC : S(State)$ is well-formed with respect to an AND-state $s$ if:

$$\text{if } s \in SC \text{ then } \forall s' \in \text{CHILD}(s) . s' \in SC$$

Well-formed Model 2.12 (OR-state Configuration)  A state configuration $SC : S(State)$ is well-formed with respect to an OR-state $s$ if:

$$\text{if } s \in SC \text{ then } \exists s' \in \text{CHILD}(s) . s' \in SC$$

Well-formed Configuration 2.1 (State Configuration)  A state configuration $SC : S(State)$ is well-formed if it is well-formed with respect to every AND-state and every OR-state.

Well-formed Model 2.13 (Well-formed Transition)  A transition $t$ is well-formed if:

- Every action in $t.a$ is well-formed
- The guard $t.g$ is well-formed
- Given a well-formed state configuration $SC : L(State)$, then:
  - $(SC \setminus \text{DEACTIVATES}(t)) \cup \text{ACTIVATES}(t)$ is also a well-formed state configuration

Well-formed Model 2.14 (Statechart)  A statechart is well-formed if:

- Every AND and OR states are well-formed
- Every transition is well-formed

Well-Formed UML Model  Well-formed Model 2.15 (UML Model)  A UML model is well-formed if:

- The class diagram is well-formed
- Every statechart diagram is well-formed
- The object diagram yields a well-formed configuration (defined on page 64).
B.1.4 Method Call Actions

Method call actions are a special case because they have to be elaborated. Hence, the trans static function is partially elaborated.

Consider the action \( a : ACall \). The model contains the OCL expression \( a.rcv \), a method name \( m \), a list of parameters \( a.exps \) and the calling type \( a.ty \). The method signature \( a.cmsig \) is computed as follows. First, the type of \( a.rcv \) is computed according to the typing rules given in Section C.1. This must be a class. Let \( C \) be that class. The method signature \( msig \) is elaborated according to the type of each parameter. The complete method signature \( a.cmsig \) is elaborated according to the \texttt{app} function, i.e. \( a.cmsig = \texttt{app}(C, msig) \). The method exists if the action is well-formed.

Dynamically, however, another method may be called. The \texttt{lookup} function is used to verify whether an overridden method is available. For example, consider the class diagram of Fig. B.1.

\[
\begin{align*}
\text{A} & \quad m &: A \rightarrow void \\
\text{B} & \quad m &: B \rightarrow void \\
\text{C} & \quad m &: B \rightarrow void \\
\text{Client} & \quad c &: C
\end{align*}
\]

Figure B.1: Method Inheritance Example

Assume that a transition of the \texttt{Client}’s statechart specifies the following action:

\[
\text{self.b.m(self.c)}
\]

This corresponds to a virtual method call. The expression providing the parameter value is of type \( B \). Hence, the extracted action is the following tuple:

\[
(\text{self.b}, (B, (m, ((B, \bot)), \bot)), (\text{self.c}), \text{VIRTUAL})
\]

Now, the set of applicable methods is computed from the \((B, (m, ((x, B)), \bot))\) method signature. This gives the following set.

\[
\{(A, (m, ((A, x)), (\bot, r))), (B, (m, ((B, x)), (\bot, r)))\}
\]

Clearly, the method defined in class \texttt{B} is more specific. The elaborated action is as follows:
(self.b, (B, (m, ((B, x)), (⊥, r))), (self.c), VIRTUAL)

Now, assume that the action is fired with the $b$ field referencing an object of type $C$. As the calling type is virtual, the LOOKUP function is used. The overridden method in class $C$ will be called.

### B.2 Configuration Vocabulary

#### B.2.1 Extracted Information

**State-Machines and Signals**

```plaintext
dynamic signals : $S($Signal$)$
dynamic actstates : Obj $\rightarrow$ $S($State$)$
```

**Values**

- `sort Int` $\equiv \{\ldots, -1, 0, 1, \ldots\}$
- `sort Bool` $\equiv \{\text{TRUE, FALSE}\}$
- `sort SimpleVal` $=$ `Int` $\mid$ `Bool` $\mid$ `Obj`
- `sort ListVal` $=$ $\mathcal{L}$(`SimpleVal`)
- `sort Val` $=$ `SimpleVal` $\mid$ `ListVal`

**Objects**

- `sort ObjEnv` $=$ `ClassField` $\rightarrow$ `Val`
- `sort Heap` $=$ $T(c : \text{Class}, env : \text{ObjEnv})$
- `dynamic heap` $:$ `Obj` $\rightarrow$ `Heap`

**Methods and Threads**

- `sort Var` $=$ `Var` $\rightarrow$ `Val`
- `sort FrameEnv` $=$ `Var` $\rightarrow$ `Val`
- `sort Frame` $=$ $T(mth : \text{ClassMSig}, env : \text{FrameEnv}, snd : \text{Obj}, rcv : \text{Obj})$
- `sort CallStack` $=$ `Frame`$^*$
- `dynamic cont` $:$ `Obj` $\rightarrow$ `CallStack`
- `dynamic returns` $:$ `Obj` $\rightarrow$ $T(msig : \text{MSig}, v : \text{Val})$

#### B.2.2 Well-Formedness

**Well-formed Configuration 2.2 (State-Machine)**  The function `actstates` is well-formed if for every object $o$, `actstates(o)` is a well-formed state configuration (defined on page 61).
Well-formed Configuration 2.3 (Well-Typed Value) Let \( v : Val \) and \( \tau : Type \), we say that \( v \) is well-typed if:

\[
\square(v, \tau) \iff \begin{cases} 
\tau = BOOL & \Rightarrow v \in \text{Bool} \\
\tau = \text{INT} & \Rightarrow v \in \text{Int} \\
\tau \in \text{CLASSES} & \Rightarrow v \in \text{Obj} \ \text{and} \ \text{heap}(v).o \leq_{h} \tau \\
\tau = \mathcal{L}(\tau') & \Rightarrow v \in \mathcal{L}(\tau) \ \text{and} \ \forall v' \in v. \ \square(v', \tau') 
\end{cases}
\]

Well-formed Configuration 2.4 (Environment) For every object \( o : \text{Obj} \) such that \( \text{heap}(o) \neq \bot \). Let \( C \) be that object’s class (i.e. \( C = \text{heap}(o).c \)). Then we must have:

\[
\forall (D, f) \in \text{FIELDS}^{*}(C). \ \text{heap}(o).\text{env}((D, f)) \neq \bot
\]

Well-formed Configuration 2.5 (Objects) Each value in the environment is well-typed.

\[
\forall o \in \text{Obj}. \ \forall cf \in \text{FIELDS}^{*}(\text{heap}(o).c) \Rightarrow \square(\text{heap}(o).\text{env}(cf), cf.ty)
\]

Well-formed Configuration 2.6 (Calling Stack) For every object \( o \) such that \( \text{heap}(o).c \in \text{THREADS} \), then \( \text{cont}(o) \neq \bot \). Furthermore, for every frame \( f \in \text{cont}(o) \), the sender and the receiver object exists, i.e. \( \text{heap}(f.snd) \neq \bot \) and \( \text{heap}(f.rcv) \neq \bot \).

Well-formed Configuration 2.7 (Method Returns) The returned value is of the correct type. For every object \( o \) such that \( \text{heap}(o).c \in \text{THREADS} \) and \( \text{returns}(o) \neq \bot \). Let \( (msig, v) = \text{returns}(o) \), then we have:

\[
\square(v, msig.r.ty)
\]

Well-formed Configuration 2.8 (Configuration) A UML model configuration is well-formed if the functions \( \text{heap}, \text{cont} \) and \( \text{returns} \) are well-formed.

B.2.3 Initial Configuration

Initial Configuration 2.1 (Active States) Initially, active states correspond to \( \text{INITSTATES} \):

\[
\text{actstates}(o) = \text{INITSTATES}(\text{heap}(o).c)
\]

Initial Configuration 2.2 (Signals) Initially, the set of active signals is empty, that is:
Initial Configuration 2.3  In the initial configuration, every thread has the following calling stack:

\[ \forall o : Obj \, .\, heap(o).c \in \text{THREADS} \Rightarrow \text{cont}(o) = \langle (\bot, \emptyset, o, o, \emptyset) \rangle \]

This is necessary for the thread to be active.

B.3 Additional Vocabulary

Temporary Configuration

<table>
<thead>
<tr>
<th>dynamic</th>
<th>( \text{signals} )</th>
<th>: ( S(\text{Signal}) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>dynamic</td>
<td>( \text{actstates} )</td>
<td>: ( \text{Obj} \rightarrow S(\text{State}) )</td>
</tr>
<tr>
<td>dynamic</td>
<td>( \text{heap} )</td>
<td>: ( \text{Obj} \rightarrow \text{Heap} )</td>
</tr>
<tr>
<td>dynamic</td>
<td>( \text{cont} )</td>
<td>: ( \text{Obj} \rightarrow \text{CallStack} )</td>
</tr>
<tr>
<td>dynamic</td>
<td>( \text{returns} )</td>
<td>: ( \text{Obj} \rightarrow T(msig : MSig, env : FrameEnv) )</td>
</tr>
</tbody>
</table>

Current Values

| dynamic   | \( \hat{\text{th}} \) | : \( \text{Obj} \) \quad \text{Current thread} |
| dynamic   | \( \hat{o} \) | : \( \text{Obj} \) \quad \text{Current object} |
| dynamic   | \( \hat{t} \) | : \( \text{Trans} \) \quad \text{Current transition} |
| dynamic   | \( \hat{a} \) | : \( \mathcal{L}(\text{Action}) \) \quad \text{Current actions} |

Iteration

| dynamic   | \( \text{iterObj} \) | : \( \text{Obj} \rightarrow \text{Bool} \) |
| dynamic   | \( \text{iterTrans} \) | : \( \text{Obj} \times \text{Trans} \rightarrow \text{Bool} \) |

OCL Expressions

| external  | \( \vdash \) | : \( \text{OclExp} \times \text{Type} \) |
| external  | \( \llbracket \cdot \rrbracket \) | : \( \text{OclExp} \times \text{OclEnv} \rightarrow \text{Val} \) |
| external  | \( \llbracket \cdot \rrbracket^{\mathcal{L}} \) | : \( \mathcal{L}(\text{OclExp}) \times \text{OclEnv} \rightarrow \mathcal{L}(\text{Val}) \) |

Processing Transitions  Notice how the \( \text{enabledSet} \) and \( \text{fireSet} \) contain couples holding an object and a transition. The object indicates the state-machine to which the transition belongs.

| dynamic   | \( \text{enabledSet} \) | : \( S(T(\text{Obj, Trans})) \) |
| dynamic   | \( \text{fireSet} \) | : \( S(T(\text{Obj, Trans})) \) |
Object Creation

\[
\text{static} \quad \text{NEWOBJ} : \text{Class} \rightarrow \text{Heap}
\]

Consistency Checks

\[
\begin{align*}
\text{dynamic} \quad \text{diff} & : \text{Obj} \times \text{ClassField} \rightarrow \text{Int} \\
\text{dynamic} \quad \text{objEnvUpdates} & : \text{Obj} \times \text{ClassField} \rightarrow \text{Val} \\
\text{dynamic} \quad \text{stepCall} & : \text{Bool}
\end{align*}
\]

End of Computation

\[
\text{dynamic} \quad \text{done} : \text{Bool}
\]

B.3.1 Well-Formedness

Note that these well-formedness constraints are re-evaluated before every step computation. This allows us to consider newly created objects, for example.

Additional Vocabulary 2.1 (Object Iteration)

\[
\forall o \in \text{Obj} . \ \text{iterObj}(o) = \text{true}
\]

Additional Vocabulary 2.2 (Transition Iteration)

\[
\text{iterTrans}(t, o) = \text{true} \iff \text{SCOPE}(t) \in \text{STATES}^*(\text{heap}(o).c)
\]

Additional Vocabulary 2.3 (Enabled Set)

\[
\text{enabledSet} = \emptyset
\]

Additional Vocabulary 2.4 (Fire Set)

\[
\text{fireSet} = \emptyset
\]

Additional Vocabulary 2.5 (Initial Object) \quad The initial object \( \delta \) is \( \bot \).

Additional Vocabulary 2.6 (Initial Transition) \quad The initial transition \( \tilde{t} \) is \( \bot \).
Additional Vocabulary 2.7 (Multiplicity)

\[ \forall o \in \text{Obj}. \ \text{diff}(o) = 0 \]

Additional Vocabulary 2.8 (Assignation Coherence)

\[ \forall o \in \text{Obj}. \ \forall cf \in \text{FIELDS}^\ast(\text{heap}(o), c). \ \text{objEnvUpdates}(o, cf) = \bot \]

Additional Vocabulary 2.9 (Step Method Call)

\[ \text{stepCall} = \text{false} \]

B.4 Transition Rule

B.4.1 Choosing a Thread

\[ \Pi_{\text{chooseThread}} \]

\[
\begin{align*}
\epsilon &: \text{Obj}, \ \text{heap}(o).c \in \text{THREADS}; \\
\Pi &: \text{th} \coloneqq o \\
\Pi &: \Pi_{\text{ES}}
\end{align*}
\]

B.4.2 Collecting Enabled Transitions

Control Flow
Iterating Objects and Transitions

\[ \Pi_{\text{selectObj}} \]

\[ \epsilon : \text{Obj} . ~ \text{iterObj}(o) \]

\[ ? \text{iterObj}(o) := \text{false} \]

\[ \Pi_{\text{toST}}(o) \]

\[ \Pi_{\text{toET}}(t) \]

\[ \Pi_{\text{acceptedTrans}} \]

\[ \Pi_{\text{rejectedTrans}} \]

\[ \Pi_{\text{enabledSet}} := \text{enabledSet} \cup \{ i \} \]

In the following, notice that a transition is iterated only if it is enabled with respect to its source state and its guard. This is verified by the \( \Phi_{\text{enabled}}(o, t) \) predicate. Notice also the \( \text{diff} \) function is reinitialized. This function allows us to verify whether the object creation and deletion actions of a transition respect the multiplicity requirements of the reference field.

\[ \Pi_{\text{selectTrans}} \]

\[ \epsilon_{1 \in \text{TRANS}} . ~ \text{iterTrans}(o, t) \land \Phi_{\text{enabled}}(\tilde{o}, t) \]

\[ ? \text{iterTrans}(\tilde{o}, t) := \text{false} \]

\[ \Sigma_{cf \in \text{FIELDS}} : \text{diff}(o, cf) := 0 \]

\[ \Pi_{\text{toET}}(t) \]

\[ \Pi_{\text{toST}}(o) \]

\[ \Pi_{\text{toSO}} \]

In the following predicate, we use the \( \text{OCL} \) expression evaluation function. Note that the variable environment \( \zeta \) is always:

\[
\{ \text{returns}(\tilde{t}).\text{msig.r.p} \rightarrow \text{returns}(\tilde{t}).\text{val} \}
\]

\[
\{ \text{SELF} \mapsto \tilde{o} \} \]
**APPENDIX B. UML SEMANTICS DETAILS**

\[ \Phi_{\text{enabled}}(o,t) \]

\[
\begin{align*}
  t.\text{src} & \in \text{actstates}(o) \\
  \land & \quad t.\text{ty} = \text{SIGNAL} \Rightarrow t.\text{tr.s} \in \text{signals} \\
  \land & \quad t.\text{ty} = \text{MCALL} \Rightarrow t.\text{tr.cmsig} = \text{top}(\text{cont}(\tilde{\text{th}})).\text{meth} \\
  \land & \quad t.\text{ty} = \text{MRETURN} \Rightarrow t.\text{tr.cmsig.msig} = \text{returns}(\tilde{\text{th}}).\text{msig} \\
  \land & \quad [t.g]_{\zeta} = \text{true}
\end{align*}
\]

**Evaluating Actions** The following rule verifies the possibility of firing a given action. A signal action can always be fired. Other actions correspond to a method’s behavior, so they can be fired only if the object is currently executing a method. The predicate \( \Phi_{\text{activeBehavior}}(o,t) \) also verifies that the transition belongs to the class defining the method. This is part of the behavior inheritance mechanism.

\[ \Pi_{\text{evalTransition}} \]

\[
\begin{align*}
  \hat{a} & \neq () \\
  \Rightarrow & \quad \hat{a} := \text{tail}(\hat{a}) \\
  \Rightarrow & \quad a := \text{hd}(\hat{a}) : \\
  & \quad a.\text{ty} = \text{SIGNAL} \\
  \Rightarrow & \quad \Phi_{\text{activeBehavior}}(\hat{o},t) \\
  \Rightarrow & \quad \Pi_{\text{evalAction}}(a) \\
  \Rightarrow & \quad \Pi_{\text{acceptedTrans}} \\
  \end{align*}
\]

\[ \Phi_{\text{activeBehavior}}(o,t) \]

\[
\begin{align*}
  \text{top}(\text{cont}(\tilde{\text{th}})).\text{rcv} = o \\
  \land & \quad t.\text{src} \in \text{STATES}(\text{top}(\text{cont}(\tilde{\text{th}}))).\text{cmsig.c})
\end{align*}
\]

The case of an assignation, object creation, or deletion calls for verification that multiplicity is not violated. This is done by computing the resulting size of a reference field upon the firing of the action and comparing it with the declared multiplicity (using the \( \Phi_{\text{violatesMult}}(s,f) \) predicate). As a transition may fire many such actions, the \( \text{diff} \) function is used to accumulate the size modifications.

\[ \Phi_{\text{violatesMult}}(s,f) \]

\[
\begin{align*}
  f.\text{mult}.up \neq \bot \Rightarrow s > f.\text{mult}.up \\
  \land & \quad f.\text{mult}.low \neq \bot \Rightarrow s < f.\text{mult}.low \\
  \land & \quad f.\text{mult}.low = \bot \Rightarrow s \leq 0
\end{align*}
\]
### B.4.3 Selecting Transitions to Fire

**Control Flow**

```
FT
buildFireSet

selectTrans

toEC(o,t)

evalConflict

rejectedTrans

acceptedTrans
```

**\(\Pi_{\text{Pi}}\)**

\[\Sigma_{o: \text{Obj}} : \Sigma_{cf \in \text{FIELDS}(\text{heap}(o), c)} : \text{diff}(\check{o}, cf) := 0\]

**\(\Pi_{\text{toEC}(o,t)}\)**

\[
\begin{align*}
\check{o} & := o \\
\check{t} & := t \\
\check{a} & := t.a \\
\end{align*}
\]

**\(\Pi_{\text{acceptedTrans}}\)**

\[
\begin{align*}
\check{o} & := o \\
\check{t} & := t \\
\check{a} & := t.a \\
\text{fireSet} & := \text{fireSet} \cup \{(\check{o}, \check{t})\}
\end{align*}
\]

**Selecting Transitions** When building the \(\text{fireSet}\), a transition is selected for addition if it is in the \(\text{enabledSet}\) and if it is not conflicting with a transition already added to the \(\text{fireSet}\) (using the \(\Phi_{\text{fireable}}(o, t)\) predicate).
Evaluating Conflict  Furthermore, the following rule determines whether adding the transition to the \( \text{fireSet} \) would create a conflict. If so, the transition is rejected. Signal actions are always accepted. Object creation and deletion actions are rejected if they produce a multiplicity violation. Again, this is done with the \( \Phi_{\text{violatesMult}}(s,f) \) predicate. However, the \( \text{diff} \) function is initialized only once, before computing the \( \text{fireSet} \). Hence, the size modifications are accumulated for each transition added to the \( \text{fireSet} \). A transition is rejected if it violates multiplicity with respect to the effects of the previously added transitions. In the case of assignation action, the function \( \text{objEnvUpdate} \) and the \( \Phi_{\text{coherentUpdate}}(o,cf,v) \) are used to verify whether the assignation is consistent with the assignations of the previously added transitions. In the case of method call and method return actions, the \( \text{stepCall} \) function is used to enforce the requirement that only one such action may be fired in one step.
\[ \Pi_{evalConflict} \]
\[ \tilde{a} \neq \emptyset \]
\[ \begin{array}{l}
\quad ? \quad \parallel \quad \tilde{a} := tail(\tilde{a}) \\
\quad a := hd(\tilde{a}) : \\
\quad \textbf{case} \ a.ty \ \textbf{of} \\
\quad \quad \text{SIGNAL} \rightarrow \\
\quad \quad \quad \circ \\
\quad \text{NEW} \rightarrow \\
\quad \quad d := \text{diff}(\tilde{a},a.cf) + 1 : \\
\quad \quad \parallel \quad \text{diff}(\tilde{a},a.cf) := d \\
\quad \quad \parallel \quad \Phi_{\text{violatesMult}}(\text{size(heap}(\tilde{a}).\text{env}(a.cf)) + d,a.cf.f) \\
\quad \quad \quad ? \quad \Pi_{\text{rejectedTrans}} \\
\quad \text{DEL} \rightarrow \\
\quad \quad d := \text{diff}(\tilde{a},a.cf) - 1 : \\
\quad \quad \parallel \quad \text{diff}(\tilde{a},a.cf) := d \\
\quad \quad \parallel \quad \Phi_{\text{violatesMult}}(\text{size(heap}(\tilde{a}).\text{env}(a.cf)) + d,a.cf.f) \\
\quad \quad \quad ? \quad \Pi_{\text{rejectedTrans}} \\
\quad \text{ASSIGN} \rightarrow \\
\quad \quad \Phi_{\text{coherentUpdate}}(\tilde{a},a.cf,[a.e]_\zeta) \\
\quad \quad \quad ? \quad objEnvUpdate(\tilde{a},a.cf) := [a.e]_\zeta \\
\quad \quad \quad : \quad \Pi_{\text{rejectedTrans}} \\
\quad \text{MCALL} \\
\quad \text{MRETURN} \rightarrow \\
\quad stepCall \\
\quad \quad \quad ? \quad \Pi_{\text{rejectedTrans}} \\
\quad \quad \quad : \quad stepCall := \text{true} \\
\quad \quad \quad \parallel \quad \Pi_{\text{acceptedTrans}} \\
\end{array} \]

\[ \Phi_{\text{coherentUpdate}}(o,cf,v) \]
\[ \begin{array}{l}
\quad \text{objEnvUpdate}(o,cf) = \bot \\
\quad \lor \quad \text{objEnvUpdate}(o,cf) = v \\
\end{array} \]

**B.4.4 Firing Transitions**

Control Flow
Needs to be translated into mathematical notation
In the following, notice how a new machine is instantiated when the object is created. This is done by setting its active states using INITSTATES.

\[\Pi_{\text{createObj}}(o, cf, C)\]

\[\partial o':\text{Obj} :\]
\[\text{ || } env := \text{heap}(\partial).env : \]
\[val := o' : env(cf) : \]
\[\text{heatmap}(\partial) := (C, env \cup \{cf \mapsto val\}) \]
\[\text{ || heatmap}(o') := \text{NEWOBJ}(C) \]
\[\text{ || actstates}(o') := \text{INITSTATES}(C) \]
\[\text{ || } \Pi_{\text{createThread}}(C, o')\]

\[\Pi_{\text{fireCallAction}}(cmsig, rcv, vals, ctxt, ty)\]

\textbf{case} \( ty \) \textbf{of}

\[\text{VIRTUAL} \rightarrow \]
\[\Pi_{\text{pushMethod}}(cmsig, msig, \text{LOOKUP}(ctxt, cmsig), vals, \partial, rcv)\]
\[\text{SUPER} \rightarrow \]
\[\Pi_{\text{pushMethod}}(cmsig, msig, \text{LOOKUP}(\text{SUPER}(ctxt), cmsig), vals, \partial, rcv)\]

In the following, notice how the initial states of an object are reactivated when a method call is pushed. This
allows the object to answer the call.

\[ \Pi_{\text{pushMethod}}(\text{msig}, c, \text{vals}, \text{snd}, rcv) \]

\[
\begin{align*}
\text{actstates}(\hat{o}) & := \text{actstates}(\hat{o}) \cup \text{INITSTATES}(c) \\
\text{env} & := \text{buildMap}(\text{msig}, \text{vals}) : \\
\text{cont}(\hat{t}h) & := \text{push}(\text{cont}(\hat{t}h), ((c, \text{msig}), \text{env}, \text{snd}, rcv))
\end{align*}
\]

\[ \Pi_{\text{popMethod}} \]

\[
\begin{align*}
\text{cont}(\hat{t}h) & := \text{pop}(\text{cont}(\hat{t}h)) \\
\text{returns}(\hat{t}h) & := (\text{a.msig}, [\text{a.exp}]_\mathcal{C})
\end{align*}
\]

### B.4.5 Applying Updates

#### Control Flow

\[ \text{applyUpdates} \]

![Control Flow Diagram]

#### Applying Updates

\[ \Pi_{\text{fireUpdates}} \]

\[
\begin{align*}
\text{signals} & := \text{signals} \\
\Sigma_{o: \text{Obj}: \text{actstates}(o)} & := \text{actstates}(\hat{o}) \\
\Sigma_{o: \text{Obj}: \text{heap}(o)} & := \text{heap}(\hat{o}) \\
\text{cont}(\hat{t}h) & := \text{cont}(\hat{t}h) \\
\text{returns}(\hat{t}h) & := \text{returns}(\hat{t}h) \\
\Pi_{\text{GO}}
\end{align*}
\]

#### Garbaging Objects

\[ \Pi_{\text{garbageObjects}} \]

\[
\begin{align*}
\Sigma_{o: \text{Obj}} & \cdot \neg \Phi_{\text{referencedObj}(o)} : \text{heap}(o) := \perp \\
\Pi_{\text{GO}}
\end{align*}
\]

\[ \Phi_{\text{referencedObj}(o)} \]

\[
\Phi_{\text{heapReference}(o)} \vee \Phi_{\text{frameReference}(o)}
\]
\[ \Phi_{\text{heapReference}(o)} \]
\[ \exists o' : \text{Obj} . o \neq o' \land \exists cf : \text{Field} . o \in \text{heap}(o').\text{env}(cf) \]

\[ \Phi_{\text{frameReference}(o)} \]
\[ \exists o' : \text{Obj}.\text{heap}(o').c \in \text{THREADS} \land \exists f : \text{Frame} . f \in \text{cont}(o') \land \\
\quad o = f.\text{rcv} \\
\quad \lor \ O = f.\text{snd} \\
\quad \lor \exists v : \text{Var} . o \in f.\text{env}(v) \]

**End of Step Computation**  The update of the auxiliary function \textit{done} indicates that the step computation is finished.

\[ \Pi_{\text{done}} \]
\[ || \text{done} := \text{true} || \]
Appendix C

OCL Semantics Details

This chapter contains details about the semantics of OCL expressions and OCL constraints. It presents the type-checking of OCL expressions and the formal semantics of the adapted $\mu$-calculus.

C.1 OCL Expressions Type System

Given an OCL expression $e$, its type is determined by recursively analyzing its subexpressions. The analysis is done according to the rules of Figure C.1. When a value, a unary, or a binary operator is reached in the analysis, its type is fetched from the predefined types listed in Figure C.2. The typing environment $\Gamma$ is used to accumulate the types of the variables. In particular, we assume it holds $\text{SELF} : \tau$, given that $\tau$ is the expression’s context, which is always a class. If the OCL expression appears in a statechart, that statechart’s class is the context. If the OCL expression appears in an OCL constraint, the class is explicitly specified.

C.2 $\mu$-Calculus Semantics

Given an ASM execution graph $\Theta = (S, \rightarrow, \mathfrak{X}_0)$, the semantics of a $\mu$-formula is presented in Fig. C.3. The function $\vartheta$ is a variable assignment that maps subsets of $S$ to $\mu$-variables. As usual, $\vartheta[X \mapsto S]$ denotes the updated assignment where $X$ is mapped to $S$ and other variables are mapped according to $\vartheta$.

Note how the semantics maps a $\mu$-formula to a set of states $([\phi]_\vartheta \subseteq S)$, which contains the states where the $\mu$-formula holds. The execution graph $\Theta$ satisfies the $\mu$-formula (written $\Theta \models \phi$) if $\mathfrak{X}_0 \in [\phi]_\vartheta$, i.e. if the initial state satisfies the $\mu$-formula when evaluated with the empty variable assignment.
Figure C.1: OCL Expressions Type System

(1) \( \Gamma, \text{exp} : \tau \vdash \text{exp} : \tau \)

(2) \( \Gamma \vdash \text{lit} : \tau \) if \( \text{lit} : d \tau \)

(3) \( \Gamma \vdash \text{exp}_1 : \tau_1, \Gamma \vdash \text{exp}_2 : \tau_2 \quad \Rightarrow \quad \Gamma \vdash \text{exp}_1 \ bop \ \text{exp}_2 : \tau_3 \) if \( \text{bop} : d \tau_1 \times \tau_2 \rightarrow \tau_3 \)

(4) \( \Gamma \vdash \text{exp}_1 : \tau_1 \quad \Rightarrow \quad \Gamma \vdash \text{uop} \ \text{exp}_1 : \tau_2 \) if \( \text{uop} : d \tau_1 \rightarrow \tau_2 \)

(5) \( \Gamma \vdash \text{exp}_1 : C, \Gamma \vdash \text{D/f} : \tau \quad \Rightarrow \quad \Gamma \vdash \text{exp}_1.\text{D/f} : \tau \) if \( C \preceq_h D \) and \( (D, f) \in \text{FIELDS}^*(C) \)

(6) \( \Gamma \vdash \text{exp}_1 : \text{L}(C), \Gamma \vdash \text{D/f} : \tau \quad \Rightarrow \quad \Gamma \vdash \text{exp}_1.\text{D/f} : \text{L}(\tau) \) if \( \tau \neq \text{L}^*(\tau') \), \( C \preceq_h D \) and \( (D, f) \in \text{FIELDS}^*(C) \)

(7) \( \Gamma \vdash \text{exp}_1 : \text{L}(C), \Gamma \vdash \text{D/f} : \text{L}(\tau) \quad \Rightarrow \quad \Gamma \vdash \text{exp}_1.\text{D/f} : \text{L}(\tau) \) if \( C \preceq_h D \) and \( (D, f) \in \text{FIELDS}^*(C) \)

(8) \( \Gamma \vdash \text{exp}_1 : \text{L}(\tau_1), \Gamma \vdash \text{exp}_2 : \tau_2, \Gamma, \text{var}_1 : \tau_1, \text{var}_2 : \tau_2 \vdash \text{exp}_3 : \tau_3 \quad \Rightarrow \quad \Gamma \vdash \text{exp}_1.\text{iterate}(\text{var}_1; \text{var}_2 = \text{exp}_2 \ | \ \text{exp}_3) : \tau_3 \)
APPENDIX C. OCL SEMANTICS DETAILS

<table>
<thead>
<tr>
<th>Val</th>
<th>(true) : (\text{BOOL})</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(false) : (\text{BOOL})</td>
</tr>
<tr>
<td></td>
<td>(\ldots, -1, 0, 1, \ldots) : (\text{INT})</td>
</tr>
<tr>
<td>Bop</td>
<td>+ : (\text{INT} \times \text{INT} \rightarrow \text{INT}) Addition</td>
</tr>
<tr>
<td></td>
<td>- : (\text{INT} \times \text{INT} \rightarrow \text{INT}) Subtraction</td>
</tr>
<tr>
<td></td>
<td>* : (\text{INT} \times \text{INT} \rightarrow \text{INT}) Multiplication</td>
</tr>
<tr>
<td></td>
<td>/ : (\text{INT} \times \text{INT} \rightarrow \text{INT}) Division</td>
</tr>
<tr>
<td></td>
<td>% : (\text{INT} \times \text{INT} \rightarrow \text{INT}) Remainder</td>
</tr>
<tr>
<td></td>
<td>&lt; : (\text{INT} \times \text{INT} \rightarrow \text{BOOL}) Less than</td>
</tr>
<tr>
<td></td>
<td>&lt;= : (\text{INT} \times \text{INT} \rightarrow \text{BOOL}) Less than or equal to</td>
</tr>
<tr>
<td></td>
<td>&gt; : (\text{INT} \times \text{INT} \rightarrow \text{BOOL}) Greater than</td>
</tr>
<tr>
<td></td>
<td>&gt;= : (\text{INT} \times \text{INT} \rightarrow \text{BOOL}) Greater than or equal to</td>
</tr>
<tr>
<td></td>
<td>== : (\text{INT} \times \text{INT} \rightarrow \text{BOOL}) Equal</td>
</tr>
<tr>
<td></td>
<td>!= : (\text{INT} \times \text{INT} \rightarrow \text{BOOL}) Not equal</td>
</tr>
<tr>
<td></td>
<td>&amp; : (\text{BOOL} \times \text{BOOL} \rightarrow \text{BOOL}) Boolean AND</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>=&gt; : (\text{BOOL} \times \text{BOOL} \rightarrow \text{BOOL}) Boolean IMPLIES</td>
</tr>
<tr>
<td></td>
<td>@ : (\text{L}(\tau) \times \text{L}(\tau) \rightarrow \text{L}(\tau)) List concatenation</td>
</tr>
<tr>
<td></td>
<td>:: : (\tau \times \text{L}(\tau) \rightarrow \text{L}(\tau)) List construction</td>
</tr>
<tr>
<td>Uop</td>
<td>+ : (\text{INT} \rightarrow \text{INT}) Unary plus</td>
</tr>
<tr>
<td></td>
<td>hd : (\text{L}(\tau) \rightarrow \tau) Head of a list</td>
</tr>
<tr>
<td></td>
<td>tail : (\text{L}(\tau) \rightarrow \text{L}(\tau)) Tail of a list</td>
</tr>
<tr>
<td></td>
<td>- : (\text{INT} \rightarrow \text{INT}) Unary minus</td>
</tr>
<tr>
<td></td>
<td>! : (\text{BOOL} \rightarrow \text{BOOL}) Logical complement</td>
</tr>
</tbody>
</table>

Figure C.2: Predefined Types for OCL Expressions

\[
\begin{align*}
[\Phi]_{\theta} &= \{ \mathfrak{a} \in S \mid [\Phi]_{\mathfrak{a}} = \text{true} \} \\
[X]_{\theta} &= \emptyset(X) \\
[-\phi]_{\theta} &= S \setminus \{\phi\}_{\theta} \\
[\phi_1 \lor \phi_2]_{\theta} &= \{\phi_1\}_{\theta} \cup \{\phi_2\}_{\theta} \\
[\phi \circ \theta]_{\theta} &= \{ \mathfrak{a}' \in S \mid \exists \mathfrak{a} \in [\phi]_{\theta} : \mathfrak{a}' \rightarrow \mathfrak{a} \in \Theta \} \\
[\nu X. \phi]_{\theta} &= \bigcup \{ S \subseteq S \mid S \subseteq [\phi]_{\theta[X \leftarrow S]} \} \\
[\phi_1 \land \phi_2]_{\theta} &= \{ \neg (\neg (\phi_1 \lor \neg \phi_2)) \}_{\theta} \\
[\phi_1 \rightarrow \phi_2]_{\theta} &= \{ \neg \phi_1 \lor \phi_2 \}_{\theta} \\
[\Box \phi]_{\theta} &= \{ \neg \phi \}_{\theta} \\
[\mu X. \phi]_{\theta} &= \{ \neg \nu X. \neg \phi \}_{\theta[X \leftarrow \neg \phi]} \\
\end{align*}
\]

Figure C.3: \(\mu\)-Calculus Semantics