OMEGA

Correct Development of Real-Time Embedded Systems

IST-2001-33522

Title :	Assertion Languages for Object Structures in UML
Author(s) :	Marcel Kyas (CAU) and Frank de Boer (CWI)
Editor :	CAU
Date :	09/01/2003
Identifier :	IST/33522/WP1.2/D1.2.1
Document Version :	5
Status :	Draft
Confidentiality :	Restricted

Abstract : We describe five languages for writing expressions, conditions, constraints, and assertions in UML models. Each of these languages is an extended subset of OCL, extended to write time constraints and to describe the architecture of a system. They are used as an expression language in models and the action language, as a constraint language describing the relation between objects of a model, or as a specification language describing the behaviour of objects and components. We describe the syntax and semantics of these languages. We define stereotypes enabling better type checking of constraints and their use and an exchange format is defined.

Docoment History

Version	*Date*	*Author*	*Comments*
5	09.01.03	Marcel Kyas, Frank de Boer	Corrections
4	20.12.02	Marcel Kyas, Frank de Boer	Corrections
3	06.12.02	Marcel Kyas, Frank de Boer	Refined Proposal, Problems of OCL
2	12.06.02	Marcel Kyas, Frank de Boer	Corrections
1	05.06.02	Marcel Kyas, Frank de Boer	Initial Proposal

1. Deliverable Introduction

This deliverable describes the syntax and semantics of our extension of OCL, which we call ASO (assertion language for object structures).

OCL and ASO reasons about the structure of the states of the system, possibly including path histories. OCL and ASO can be used as a general expression language within the Omega UML language. Whenever assertions on the structure of states and its history are needed in an UML model, they can be expressed in ASO. OCL may be used to specify the internal and observable behaviour of objects and the internal and observable behaviour of components. For the specification of observable behaviours we use an assertion language on the history of observable events. The internal behaviour is specified using sequences of states.

Since the first mile stone, ASO has been redesigned to improve compatibility with OCL 2.0. Most constraints expressed in OCL are constraints in ASO. The language constructs of ASO contain the events and time mechanisms described in deliverable D1.1.2 dealing with the real-time extension of UML, for completeness and selfcontainedness of the document.

We have analysed possible type systems for OCL and their extension. Despite our initial concerns with OCL's type system, we have decided to use a solution that is compatible with current object oriented programming languages, because other solutions cause more problems than they solve.

Our variant of OCL does not allow specification of liveness properties; to do this, OCL may be used in protocol state machines and live sequence charts.

Assertion Languages for Object Structures in UML

Marcel Kyas
Frank S. de Boer

January 9, 2003

Contents

Chapter 1

Introduction

In this report we describe the abstract syntax and semantics of our assertion language for object structures in the *unified modelling language* (UML). In particular we will

- We define an extension of the *Object Constraint Language* (OCL) for requirements specification of architectures.
- We define a formal semantics which will be a basis for verification (see work package 2.2 in [32]).
- We define a package of data types for constraining class diagrams, for behavioural specifications, and for requirements specification of architectures.
- We define a concrete syntax for tool support and define an interchange format within the XMI.

During the development of the extension of OCL we discovered, that OCL has a number of features which do not serve our purposes. These are described in Chapter 2. Our language removes most of these features. To avoid confusion, we therefore call our modified and extended version of OCL an *assertion language for object structures* (ASO). In this report we describe the syntax and semantics of ASO.

In this chapter we define the purpose and the context of ASO. We describe the structure of this report and the conventions used.

1.1 Relation to the Omega Project

Omega aims at the definition of a development method in UML for embedded and real-time systems based on formal techniques used to improve commercially available UML tools. In this report we define a formal language for requirements specifications of architectures. These requirements are defined in UML class diagrams expressed in the *Omega kernel language* as defined in [12]. The main differences to [12] are:

- The description of a model is based on the UML meta model, as described in [2]. Our formalism is by this more abstract than the one presented in [12].

- We have integrated the notion of time and event from [18]. This step enables us to use a proof method similar to the one presented in [20].
- We have defined a syntax for the interchange format proposed in Omega [28]. A second format is also described, similar to the format of OCL constraints.
- Both ASO and OCL are languages about state properties. It is similar to assertions in Hoare style verification. We have added a history variable for events which is automatically updated. This could also be achieved by augmenting the original program. For lifeness properties we suggest using Live Sequence Charts [23] or Protocol State Machines.

1.2 Relation to the Unified Modelling Language

The description of the formal semantics of the UML and ASO uses the UML meta-model as defined in [1] and [2]. We considered the UML™ Profile for Schedulability, Performance, and Time Specification [31] for our model of time. Our work started as an extension of OCL 1.4 [30], then we changed to OCL 2.0 [5], but finally we found that OCL had many properties which make its use for machine-aided verification difficult. These issues are addressed in Chapter 2. Instead, we have decided to create a language which is similar to OCL, but breaks the syntax, wherever the semantics of OCL and our language are different.

1.2.1 UML Class Diagrams

UML class diagrams define the structure of a system and thus define the variables and signatures of an ASO formula. This implies that each ASO formula is connected to a class diagram in that an assertion is always evaluated in the context of instances of a class diagram. A model often consists not of only one class diagram but of a collection of class diagrams, each of them modelling a specific aspect of the model. In each model different aspects of the used classes are shown. For example, the definition of certain classes of the UML meta model is spread over a collection of class diagrams, each of them showing the information relevant to the model. The normative text[1] on the semantic of a class diagram collects the members defined in each class diagram.

Our assertion language is able to handle this aspect specific modelling. The semantics of the assertion language allows assertions on incomplete class diagrams (in the sense that not all members of a class are shown in a class diagram). We hope to suggest methods which will help to identify inconsistencies that arise from the specification of a class in different class diagrams.

1.2.2 UML State Machines

The *behaviour* of a system in UML is modelled using UML state machines. Two kinds of state machines are defined in UML: state machines and protocol state

[1]The normative text is the (formal) text *defining* the standard. In case of doubt it supersedes all other explanations of the standard text.

machines. The semantics of state machines in the OMEGA kernel language is defined in [12].

We propose to use protocol state-machines as a visual formalism for specifying the set of all legal communication histories. The study of protocol state machines is outside the scope of this report.

1.2.3 Sequence Diagrams

The UML sequence diagrams are replaced by life sequence charts in this project.

Live sequence charts are a suitable formalism for specifying liveness properties of objects. Therefore we restrict ASO to the specification of safety properties.

1.2.4 The Object Constraint Language

Our specification language for architectural descriptions is based on OCL version 2.0 [5]. Event though we have identified many inadequacies of OCL, we have decided to use a conservative extension of OCL. This means that most existing OCL constraints are still usable as ASO constraints. The concrete syntax of the language is the same whenever the semantics of the expressions are the same. Existing models may be converted to our formalism without any change and still have the same meaning. Some constraints will have a different meaning in our language, because we have decided to use a two-valued interpretation of constraints. In these cases we changed the syntax, to make the differences apparent.

OCL was not designed to be a specification language, and it does not work well as a constraint language. Different design decisions during the development of OCL make it closer to a query language like SQL and OQL [10, 35].

Chapter 2 discusses the problems we have identified in OCL and how we intend to fix them. Chapter 4 defines the abstract syntax of our assertion language and points out the syntactic differences. Chapter 5 describes the semantics of ASO and summarises the semantic differences of our assertion language.

1.3 Design Aims

The language we propose is designed to achieve the following goals:

- The assertion language defined in this report should not interfere with the many UML-based design methods. Instead, we want to define a precise specification language, which can be integrated seamlessly into these design methods. Hence, we have to take the following methods into account:
  - Abstraction and refinement are used in many UML-based methods. Very often a class is refined into a collection of classes; then an object may be refined into a collection of objects. See for instance [14].
  - Aspect orientation: Systems are commonly modelled by a collection of diagrams, depending on the desired functionality. We do not believe that someone will describe an implementation of a CORBA component written in Java with an UML diagram, because a vast number of classes and methods are involved only to access the CORBA

services. Instead, a collection of class diagrams is used to describe different aspects of these classes. As a consequence, we have to take into account, that a class diagram may lack information which may appear in the implementation.

- UML Class Diagrams are not well suited to describe the dynamic object structures of object-oriented programs. Object diagrams, a kind of class diagrams, can be used to illustrate snapshots of object structures. Hence, a formula will be evaluated in the context of an object structure.

- The language should support operational specifications in the style of Eiffel [26, 27]. Each class is equipped with an invariant Each method is described in terms of pre- and postconditions. These are two predicates relating initial states of a method call to its final states.

- A history mechanism is defined, with a language to reason about histories.

Our specifications are split into two levels. The *local* specification language is used to define properties of the local state of an object. The local state of an object is given by the valuation of all its instance variables and the valuation of a methods parameters. On the local level it is not possible to dereference the state of other objects. The *global* specification language is used to do this. It is used to specify properties of the object structure.

ASO differs from OCL in the following ways:

- ASO is an assertion language for object structures. Our assertions are intended to describe properties of object structures. This is a subset of the intended functionality of OCL 2.0.

- OCL as published in [30] does not have a clear formal semantics. The situation is changing in OCL 2.0, where a formal semantics for OCL is currently being developed. We do not want to wait for the completion of this task.

- OCL does not distinguish between local and global specifications. The distinction between a local and a global assertion language adheres to the principle of information hiding and supports a compositional description of the behaviour of a system of objects. Also, the lack of this distinction can be used to violate the information hiding principle: It is possible to define class invariants, which make assumptions about the state of other objects using the navigation mechanism. As a result, we can give implementations, where a local change of an object violate the class invariant of another object. An example is given in [3].

- OCL is a three-valued logic. This is not adequate for an assertion language: What exactly is the meaning of a state formula, whose value is undefined? Our assertion language is two-valued, but it contains special artifacts for undefined values of expressions only. No boolean expression will have an undefined value.

1.4 The Semantic Model

Part of the specification of the system are assertions. Another part of it are UML class diagrams, which give a *structural description* of a system. The structural description comprises the instance variables, methods, and typing information. The class diagram also conveys other safety properties, which will be explained in Chapter 3.1.3.

- A class diagram does not specify behavioural aspects of an instance of any class appearing in it.
- Many structural aspects of object structures cannot be specified in class diagrams alone. A logical formalism has to be used to define constraints on object structures.

Object structures are the fundamental semantic entities of this report. Object structures will be formalised and discussed in Section 3.1.4.

1.5 Roadmap

In Chapter 2 we collect the problems of the syntax and semantics we have found with OCL. We motivate what needs to be changed, and why it has to be changed. This will motivate a language different from OCL.

Chapter 3 contains a description of a typed logic with subtypes. This logic is the formal foundation of our assertion language for object structures. We describe the mathematical notation used in this report, define the type system of the logic, describe the formal syntax and semantics of the logic, and relate it to object structures.

Chapter 4 defines the abstract syntax of the assertion language. The scope of this chapter is similar to the OCL "meta model." Instead of expressing the "meta model" in UML we use conventional abstract grammar. We do not define the different levels of assertions within the abstract syntax. Only the means to declare the intended level of assertions or expressions are provided. Chapter 5 describes the formal semantics of the assertion language defined in Chapter 4.

Chapter 6 describes the formal semantics of the data types which we assume to be present in the assertion language. Here we introduce the formal semantics of the data types commonly used in programming languages, define collection types like sets, bags, and sequences, and define *histories of communication events*.

Chapter 7 describes the pragmatics of using our assertion language and gives an example on how to specify with ASO. Chapter 8 gives a final conclusion and describes future work.

Appendix A defines the concrete syntax of our assertion language in BNF. We also define the interchange formats of constraint language.

We have tried to provide a comprehensive index for this report. If a page reference is set in **bold face**, then it refers to the definition of the entry. Page references set in Roman font point to instructive examples.

Furthermore, we used the following typographic conventions: Names of modelling entities appearing in the UML Standard Document are written in Sans Serif type face. Names defined in the concrete syntax of the assertion language

are written in `tele type` type face. Names defined in the semantic domain of each of these languages are written in *italic* type face.

Acknowledgements

Willem-Paul de Roever has commented on many versions of this report and gave valuable hints on references used in this report.

Martin Steffen has helped to clarify many items of this report and was always available for discussions.

Chapter 2

OCL Issues

This chapter describes the problems we have with the Object Constraint Language (OCL) and why we need a different specification language. OCL is not perfect. Indeed, many issues have been identified in the OCL, see for example [38, 10, 36, 6]. We list the most prominent issues, which are neither changed in [5] nor in [35], which is the basis for Appendix A of [5].

These issues were essentially the result of a series of design decisions. These decisions made OCL unsuitable for formal specification and verification. These design decisions also invalidated some of the design aims of OCL. According to [10] OCL's most important features are:

- Tight integration with the UML notation
- Pure specification language without operational semantics
- Basic types like Boolean, Integer, Float
- Container types Collection, Bag, Sequence, and Sets with appropriate operators
- The full type system also includes the types induced by class definitions within UML
- Navigation expressions to navigate along associations with various multiplicities
- Boolean operators to allow full propositional logic
- Existential and universal quantifiers over existing sets of objects
- Specifications are (to a large extend) executable and can therefore be used as assertions when animating the UML diagrams or generating code

OCL is a specification language that tries to mediate between the practical users needs and the theoretical work that has been done in that area.

Our discussion of OCL starts with a short critique of OCL's type system.

2.1 Type System

OCL's type system is defined on top of UML's type system [5, 2]. Types in OCL are indeed classes of object oriented programming languages. Because of this,

many specifications which make sense from a mathematical point of view are invalid in OCL; and algebraic laws do not hold anymore. To have a checkable type system, the specifier has to frequently use explicit *type coercions*. This leads to some usability issues.

2.1.1 Object-Oriented Type System

First, the type system of OCL is based on type systems of object-oriented programming languages. This results in some annoying restrictions. Among others, the union operation on sets is not commutative anymore.

Example 2.1.1. The expression $3.0 + 4$ is well typed, but $4 + 3.0$ is not.

This example mainly breaks, because the type Integer is a subtype of Real, and both classes define an operation + with signature $\mathsf{Integer} \times \mathsf{Integer} \to \mathsf{Integer}$ and $\mathsf{Real} \times \mathsf{Real} \to \mathsf{Real}$, but does not decide which one to use. The inheritance of UML on the other hand, mandates that the latter signature has to be taken.

Example 2.1.2. If $\theta_1 \preceq \theta_2$ and s_1 is of type θ_1 and s_2 is of type θ_2, then the expression $s_2.union(s_1)$ is well defined and returns a collection of type $\mathsf{Set}[\theta_2]$, but $s_1.union(s_2)$ is not well typed, because $Set[\theta_1] \preceq Set[\theta_2]$ and the result would be of type $Set[\theta_2]$.

This basically means that many operations lose their algebraic properties. We may conclude, that $s_2.union(s_1) \neq s_1.union(s_2)$.

2.1.2 Navigation and Types

Navigating an association with multiplicity zero or one results in an object with two types: The type of the object at the association end and a (singleton) set of this type. The use of the accessor operators (. or ->) decides which type is used. However, using strong typing becomes difficult in this setting.

Example 2.1.3. If manager is an association with multiplicity zero or one to an object of type Manager, then the expression self.manager is an object of type Manager. Writing self.manager->size() coerces this object to the type Set(Manager).

2.1.3 Parameterisation

UML provides the linguistic means to define parameterised classes and dependent types. But the formal semantics are not defined for these entities. Especially, there are no rules defined which relate these concepts with generalisation.

OCL uses parameterised classes too, and defines a type system for these. However, it does not generalise to the classes of a UML diagram.

OCL does not allow bounded parameterisation, which makes many constraints overly verbose.

Example 2.1.4. Instead of declaring a type with $s : \mathsf{Set}(T <: \mathsf{Ordered})$ to assert that all elements of the set can be ordered, one has to write constraints like

$$s\text{->}forall(e \mid e.oclIsType(\mathsf{Ordered})\ and\ \varphi(e))$$

to make use of this property.

2.1.4 Inheritance

The OCL standard by itself does not define how to constraints and sub-classing work together.

The UML standard mandates, that a subclass has to satisfy the union of all constraints defined in its super-classes (see [30, p. 2-70]):

> The constraints on the full descriptor ore the union of the constraints on the segment itself and all of its ancestors. If any of them are inconsistent then the model is ill formed.

This restricts sub-classing to (behavioural) sub-typing. Software developers often use sub-classing and overriding to change the behaviour of classes.

2.1.5 Access Modifiers

OCL does not take access modifiers like private, protected, and public into account. Constraints are written in a way that all features of a class are considered to be public. This is not a real problem. But the specifier has to exercise extra discipline, if his specifications are to be useful for clients of an API.

2.2 Semantics

OCL allows the use of operations in specifications. This may lead to the following problems:

- It adds to the operational flavor of OCL. The execution of an operation may be nonterminating or result in undefined. In these cases the expression containing this operation is said to be undefined. It is in general undecidable, whether such an operation is nonterminating, which adds to the complexity of the assertion language.
- Consider an operation applied to a collection of objects that are instances of some class C. Some elements of the collection may be of instances of one of Cs subclasses, say D. If D redefines this operation, it is not defined which operation has to be applied to each object of the collection. Even if we assume "late-binding", the meaning of all constraints may change during the evolution of the model, just by adding new subclasses which override the method used.

  By this mechanism, it is even possible to invalidate constraints in a model, in that a subclass introduces a side-effect into the operation used for a specification.

2.2.1 Ill-defined Three Valued Logic

OCL is a three-valued logic, which adds a truth-value "undefined" to the traditional truth values "true" and "false". More importantly, OCL allows any expression to have an undefined value. However, the standard does not give a hint on what the meaning of such an undefined constraint is. This creates some pitfalls for the modeller.

If OCL is used as an assertion, the standard silently assumes, that the result of an OCL expression is either true or false. In Chapter 7 of the OCL 2.0 submission version 1.5, it is only defined, when an invariant, precondition, and postcondition has to hold. Otherwise the model is considered to be "ill-formed". By this, we can conclude, that if a constraint is undefined when used as an invariant, precondition, or postcondition, then the diagram is ill-formed.

OCL expression may also appear as the initial value of an attribute or as a guard in a state machine. In these cases, the value needs an explanation. The standard offers no interpretation for this value. We introduce two possible explanations:

Exceptional Termination [1] The evaluation of the expression would raise an exception or return an arbitrary value. Examples of such situations are dereferencing a NULL pointer and division by zero. If the evaluation of the expression results in a method invocation, whose precondition or postcondition are violated by the parameter values, then we may also encounter an exception.

Divergence The evaluation of the constraint does not terminate, but results in an infinite computation. One example of such an expression would be a query operation resulting in the execution of an infinite loop.

This also means, that in addition to first and final states, we also need a state modelling *non-termination*. We define non-termination as *divergence*, that is, the computation does not terminate, and *abnormal termination*, that is, the computation aborts with an exception. We usually say, that the value of a non-terminating computation is *undefined*. Conversely, this should be the sole purpose of *undefined*.

A three-valued logic is mainly useful if you want to reason about total correctness of programs. In contrast to partial correctness, we call a program totally correct, if it satisfies its specification and, if the precondition is satisfied, is guaranteed to terminate.

The formal semantics of OCL is based on a three-valued logic with strict semantics. Essentially, a three-valued logic usable for program verification should satisfy the following conditions:

1. It defines negation as a monotone function on the flat cpo

$$\mathbb{B}_\perp \stackrel{def}{=} \{true, false, \perp_{\mathbb{B}}\} \quad , \tag{2.1}$$

   and conjunction, disjunction, and implication as monotone functions from $\mathbb{B}_\perp \times \mathbb{B}_\perp \to \mathbb{B}_\perp$. Restricting these functions to $\mathbb{B} \times \mathbb{B} \to \mathbb{B}$ gives the classical logical connectives.

2. The conjunction $\wedge$ satisfies the equation

$$[\![\mathsf{false} \wedge \mathtt{undef}]\!] = [\![\mathtt{undef} \wedge \mathsf{false}]\!] = false \quad , \tag{2.2}$$

   and by this also

$$[\![\mathsf{true} \vee \mathtt{undef}]\!] = [\![\mathtt{undef} \vee \mathsf{true}]\!] = true \quad . \tag{2.3}$$

[1] This is sometimes called *abnormal termination*.

3. Existential quantification should be equivalent to disjunction, and universal quantification should be equivalent to conjunction. By this, the reasonable predicate

$$\exists x. \frac{1}{x} = 1 \tag{2.4}$$

is equivalent to

$$\bigvee_x \frac{1}{x} = 1 \tag{2.5}$$

and denotes *true* instead of $\perp_{\mathbb{B}}$, as a strict semantics would require.

These conditions are met by Kleene's logic [21], and the conditions first two conditions are reflected in the definition of standard OCL. Table 2.1 summarises the semantics of boolean connectives satisfying conditions 1 and 2.

a	b	not a	a and b	a or b	a implies b
false	*false*	*true*	*false*	*false*	*true*
false	*true*	*true*	*false*	*true*	*true*
false	$\perp_{\mathbb{B}}$	*true*	*false*	$\perp_{\mathbb{B}}$	*true*
true	*false*	*false*	*false*	*true*	*false*
true	*true*	*false*	*true*	*true*	*true*
true	$\perp_{\mathbb{B}}$	*false*	$\perp_{\mathbb{B}}$	*true*	$\perp_{\mathbb{B}}$
$\perp_{\mathbb{B}}$	*false*	$\perp_{\mathbb{B}}$	*false*	$\perp_{\mathbb{B}}$	$\perp_{\mathbb{B}}$
$\perp_{\mathbb{B}}$	*true*	$\perp_{\mathbb{B}}$	$\perp_{\mathbb{B}}$	*true*	*true*
$\perp_{\mathbb{B}}$	$\perp_{\mathbb{B}}$	$\perp_{\mathbb{B}}$	$\perp_{\mathbb{B}}$	$\perp_{\mathbb{B}}$	$\perp_{\mathbb{B}}$

Table 2.1: Semantics of OCL's Boolean Connectives

One unfortunate fact on OCL is, that it mixes the truth value of a constraint with the fact whether the constraints definition is computable or not by a certain interpreter. This will impose many challenges for implementers. Consider formula (2.4), which may be written as

```
Integer.allInstances()->exists(x|1/x=1)
```

in OCL. Because the expression `Integer.allInstances()` does not denote a finite set, the value of the constraint is undefined, even though a satisfying value for `x` can be found by symbolic, algebraic rewriting of the constraint. This also demonstrates that OCL does not satisfy condition 3.

Example 2.2.1. Both expressions $\forall i : i \in \{0, 1\} \rightarrow \frac{i}{i} = 1$ is undefined, as is the negation $\exists i : i \in \{0, 1\} \wedge \frac{i}{i} \neq 1$.

The OCL standard mandates that each non-terminating operation call returns the value undefined. This is in general not decidable, which makes it impossible for tool vendors to implement the semantics of OCL (see also Section 2.2.6). In combination with the strict evaluation strategy, all statements over infinite sets are undefined.

Example 2.2.2. In OCL, the tautology $\forall i.i \in I \rightarrow i = i$ is *undefined* for all infinite sets I.

At last, the presence of undefined constraints and their use raises the question what it means to satisfy undefined. This is not answered.

Example 2.2.3. Obviously, any terminating program implements the post-condition $\forall i.i \in I \rightarrow i = i$, but this constraint is undefined.

Within the theory we have described here, we will show that the following formula does not have a meaning in our three valued logic:

$$\forall x(\frac{1}{x} = 1 \rightarrow x = 1) \tag{2.6}$$

By propositional reasoning and condition 3 we can convert the formula to

$$\bigwedge_x(\frac{1}{x} \neq 1 \vee x = 1)$$

for which the term $\frac{1}{x} \neq 1 \vee x = 1$ is true if $x \neq 0$ and undefined for $x = 0$. Hence the interpretation of (2.6) is $\perp_{\mathbb{B}} \wedge \mathsf{true}$, which is undefined.

The solution to this problem is to define equality as *strong equality*.

2.2.2 Equality

OCL does not define the notion of equality to use.[2] If one writes $a = b$, it is not clear how to interpret this constraint. We have the following choices:

- a and b are of the same type and all their attribute values are equal. This is called structural equality.
- a and b are identified by the same unique object identifier. This is called reference equality.

Additionally, we have to choose how we will handle undefined. Again we may choose between:

- Strong equality, where $\underline{\perp} = \underline{\perp}$ is true.
- Weak equality, where $\underline{\perp} = \underline{\perp}$ is undefined.

The OCL 2.0 standard only decides the latter question, while the first question is still open.

Strong equality is defined as:

$$[\![x = y]\!] = \begin{cases} \mathit{true} & \text{if } [\![x]\!] = [\![y]\!]. \\ \mathit{false} & \text{otherwise.} \end{cases} \tag{2.7}$$

By this we have $[\![\perp_{\mathbb{B}} = \perp_{\mathbb{B}}]\!] = \mathit{true}$ and $[\![\perp_{\mathbb{B}} = 1]\!] = \mathit{false}$.

We strongly believe that a good assertion language possesses at least a strong equality operator; it allows us to conveniently solve the above problem, and also provides the most precise notion of equality of which all others can be defined.

Remark 2.2.4. If you introduce strong relations, in which $[\![R \ni \underline{\perp}_{\mathbb{B}}]\!] = \mathit{false}$ holds, the logic becomes a two valued first-order logic. This can be extended to a class logic, either by following Oberschelp, or by introducing the axioms of Zermelo and Fraenkel.

Additionally, equality is defined as an operation on OclAny, and it does not exclude the possibility of overriding this operation.

Example 2.2.5. Assume that class C overrides $=$ to mean equality on all attributes except one, say a. Then it is not guaranteed, that $a = b$ implies $a.a = b.a$.

[2] In [10] an answer to this question is proposed.

2.2.3 Fix-point Semantics

OCL allows the definition of recursive predicates and functions, but their fix-point semantics is not precisely defined [30, p. 6-13]:

> The right-hand-side of this definition may refer to the operation being defined: that is, the definition may be recursive as long as the recursion is not infinite.

In general it is not decidable whether a recursive is infinite. Also, the meaning of such an expression is only defined if the recursion is not infinite.

Example 2.2.6. Consider the specification:

```
context number.M(n: Integer): Integer
pre: n = self.M(n)
post: result = self.M(n)
```

What is the interpretation of this specification? Is M the undefined function? Or is this specification illegal? Or is M denoting the identity function? And under which conditions may M be called?

This mixes declarative semantics, which were a design goal of OCL, with operational semantics. Also, it is not decidable, whether a fix-point can be reached in a finite number of iteration steps, but sometimes a fix-point can be deduced without knowing the number of iteration steps. Such a solution must not be used, unless one has proven that the number of steps to reach this fix-point is finite.

Such a requirement is to be too strong for a specification language.

2.2.4 Flattening

Due to the flattening in OCL there is no distinction between non-deterministic functions and function returning sets. A non-deterministic function returning sets has the same signature as a deterministic function returning a set: both return set.

OCL defines the semantic convention that each set containing an undefined value has to be undefined. As a consequence, the following statements are all true:

$$\underline{\bot} \notin \{\underline{\bot}\}$$
$$\{\underline{\bot}, X\} = \{\underline{\bot}, Y\}$$
$$head(X, \underline{\bot}) = \underline{\bot}$$

The flattening operations are not worked out well. In OCL 1.4 it is not specified, how to flatten $(\{1, 2, 3\}, \{3, 4, 5\})$. Only the explicit flattening mechanisms of OCL 2.0 give the user some control over these operations by providing an `orderedBy` operation converting sets to sequences.

2.2.5 The Meaning of @pre

OCL does not define an exact meaning of the @pre postfix-operator. The intended meaning is the value of an attribute, variable, or method in the preceding

state. But OCL does not formalise the notion of a state. Because the notion of a state also defines the notion of an observation, it strongly depends on the computational model used to formalise the behaviour. If we use methods written in a programming language, then we have a notion of a state at the invocation time and at the termination of a method, but if we use state machines too, then the observable states may be only the stable ones. The computation between two stable states may contain many operation calls, for which these intermediate states are not available.

2.2.6 Executability

Finally, we want to analyse the notion of "executability" of OCL. The language has got an operational flavor, in that truth values of expressions dependent on the fact whether the evaluation of these expressions is terminating. This is essentially done to the benefit of implementers. We argue that such an endeavour is very dangerous for users and tool vendors.

First, we note that you generally cannot decide, whether the evaluation of an expression will eventually terminate. If you consider the expressive power of OCL [8], then we cannot hope to devise an algorithm, which will decide this problem for OCL.

Given this fact, we will analyse the differences between the logical connectives in OCL and the logical connectives in a programming language. We will use the Java programming language [17] as an example.

The semantics of Java's logical or operator is a "short-circuiting" one, evaluating the sub-terms from left to right. By short-circuiting we mean that once the truth value of the expression cannot change, the remainder of the expression will not be evaluated. This results in the following truth table:

`a`	`b`	`!a`	`a && b`	`a || b`
false	*false*	*true*	*false*	*false*
false	*true*	*true*	*false*	*true*
false	$\bot_{\mathbb{B}}$	*true*	*false*	$\bot_{\mathbb{B}}$
true	*false*	*false*	*false*	*true*
true	*true*	*false*	*true*	*true*
true	$\bot_{\mathbb{B}}$	*false*	$\bot_{\mathbb{B}}$	*true*
$\bot_{\mathbb{B}}$	*false*	$\bot_{\mathbb{B}}$	$\boxed{\bot_{\mathbb{B}}}$	$\bot_{\mathbb{B}}$
$\bot_{\mathbb{B}}$	*true*	$\bot_{\mathbb{B}}$	$\bot_{\mathbb{B}}$	$\boxed{\bot_{\mathbb{B}}}$
$\bot_{\mathbb{B}}$	$\bot_{\mathbb{B}}$	$\bot_{\mathbb{B}}$	$\bot_{\mathbb{B}}$	$\bot_{\mathbb{B}}$

Table 2.2: Semantics of Java's Boolean Connectives

Apparently, if the evaluation of the first argument `a` of an expression in Java is non-terminating, then the whole expression is non-terminating, despite the value of of the second argument `b`. The reason for this is, that we cannot determine the truth value of an expression without evaluating it. But because the evaluation proceeds from left to right, we are stuck during the evaluation of `a`. By this the semantics of the boolean connectives differ from OCL; we have boxed the differing values.

Similar statements can be made for all programming languages. In programming languages, the value of an boolean expression also depends on the order

in which its arguments are evaluated. This leads to different results in different implementation languages, and even in different implementations (for example C).

Because of the undecidability of the halting problem, we cannot implement the semantics required by OCL, because we have to *choose*, in which order the arguments are evaluated, and by this we have to deviate from the truth table given in Table 2.2.

2.3 Usability

OCL expressions are at times unnecessarily verbose due to the following:

The sub-classing mechanisms and strong typing of OCL leads to frequent explicit type checking (`oclIsTypeOf`) and type coercions (`oclAsType`).

OCL is not suited to detailed design. Especially the specification of abstract data types can become lengthy and scattered among different classes implementing the data type. Sometimes it will even describe an implementation of the operation to be specified. This is very often due to the operational nature of OCL.

The use of @pre is very complicated. If one wants to refer to the previous value of an navigation expression, one has to adorn each component of the expression with @pre.

Example 2.3.1. The previous value of the expression $\mathsf{self}.a.b.c.d.e$ is not

$$\mathsf{self}.a.b.c.d.e@pre$$

or

$$(\mathsf{self}.a.b.c.d.e)@pre$$

but

$$\mathsf{self}.a@pre.b@pre.c@pre.d@pre.e@pre$$

2.4 Summary and Conclusion

This chapter collects some of the issues in OCL. Fixing some of them makes the assertion language incompatible with OCL. To this end, we will use different syntax where the semantics of the assertion language proposed in this document differs from OCL. This is to save the users of unexpected results.

Also, for sake of simplicity, we decided to not change the following issues:

- While the object-oriented type system violates some natural mathematical laws, we do not impose too many restrictions on the models to be annotated with our assertion language. As described in [36], we have to impose that the generalisation hierarchy described by the class diagrams form a complete lattice. This cannot be guaranteed in the presence of multiple inheritance. The users will probably not accept this restriction. Instead, our type system is based on a "best effort" for deducing the type.
- We have decided to provide a flattening navigation operator for unordered collections. This is closer to data base languages like SQL. In addition, we changed the usual navigation operator to a non-flattening one.

- Except for collection types we do not allow parameterised classes in the class diagram.
- We follow the proposal of UML 2.0 [2] on the issue of inheritance of constraints. Thus the user is allowed to override constraints; we are aware on the implications for verification and add precise run-time type information for specification.
- The semantics of our assertion language is changed to a two-valued logic.
- We add fix-point operators in the future.
- We do not guarantee executability.
- We change the syntax of OCL to make expressions simpler and more readable. We will preserve as much of OCL as possible.

Chapter 3

Logical and Mathematical Foundations

This chapter describes the formal foundations of the remainder of this report. It serves as a reference to the reader and may skipped during first reading. We present our notation for mathematical concepts used in the description of the formal semantics of the assertion language.

We recall the formal syntax and semantics of a multi-typed logic. This logic serves as the meta-formalism of the semantic description given in the following chapters. The formalism is standard; we have included it as a reference to readers who are not familiar with our notation. The notation used here is based on [29] and [7].

Section 3.1 describes the formal model used in this report. This formal model is based on many-sorted predicate logic, which we denote by LP. Many-sorted theories do not handle sub-typing well. To handle this, we refine the language with a type theory. This theory is introduced in Section 3.3. Section 3.4 concludes this chapter with a short summary and future work.

3.1 Formal Model

The formal model we use to interpret assertions is based on first-order structures. Our first order structures are essentially many-sorted algebras over a theory derived from an UML class diagram. We roughly follow the ideas of *membership equational logic* [25]. Other ideas were taken from [33].

The model is general enough to handle different levels of abstraction within one framework, similar to *abstract state machines* [19].

- The state and the behaviour of each individual object.
- The states and the behaviour of a set of objects.

3.1.1 Formal Languages

The meta-language we will use to describe object structures is *many-sorted logic*. A many-sorted language consists of a set of sorts and a signature. Let S be a set whose elements we call sorts and let S^* denote the free monoid over S with

unit ϵ. We will write $S^+ \stackrel{def}{=} S^* \setminus \{\epsilon\}$. A signature is of the form $f : w \to s$, where $w \in S^*$ and $s \in S$, or of the form $r : w$, where $w \in S^+$.

A term with the signature $w \to s$ is called a *function*. If this term has the signature $\epsilon \to s$, then it is called a *constant*. A term with the signature $w \in S^+$ is called a *relation*.

A language is defined by

- A set $\mathcal{R}$ of *relation symbols*.
- A set $\mathcal{F}$ of *function symbols*.
- A *signature function* σ assigning each member of $\mathcal{R}$ a string $w \in S^+$ and each member of $\mathcal{F}$ a string $w \to s$, where $w \in S^*$ and $s \in S$.

If $\mathcal{R} = \emptyset$, then we call the language *algebraic*, if $\mathcal{F} = \emptyset$, we call it *relational*. We can now define the syntax of a many-sorted language.

Definition 3.1.1 (Terms of $LP(S, C, \sigma)$). Let X be a set of variables and Γ a function $X \to S$. The language of *terms* of $LP(S, C, \sigma)$ is defined by:

1. Each variable $v \in X$ is a term of sort $\Gamma(v)$.
2. If $t_1, \ldots, t_n$ are terms of sort $s_1, \ldots, s_n$ and f is a term of sort $s_1 \cdots s_n \to s$, then $f \circ (t_1, \ldots, t_n)$ is a term of sort s.

Definition 3.1.2 (Formulae of $LP(S, C, \sigma)$). We define the language of *formulae* of $LP(S, C, \sigma)$ by:

1. The symbols **true** and **false** are formulae.
2. If t_1 and t_2 are terms of sort s, then $t_1 = t_2$ is a formula.
3. If $t_1, \ldots, t_n$ are terms of sort $s_1, \ldots, s_n$ and R is a constant with $\sigma(R) = s_1 \cdots s_n$, then $R \ni (t_1, \ldots, t_n)$ is a formula.
4. If φ and ψ are formulae, then $\neg\varphi$ and $(\varphi \wedge \psi)$ are formulae.
5. If φ is a formula and v is a variable, then $(\exists v.\varphi)$ is a formula.
6. There are no other formulae.

Definition 3.1.3 ($\mathcal{L}$-Structure). An *$\mathcal{L}$-structure* is defined by two functions:

1. A function $\mathcal{D} : S \to A$, which assigns to each sort $s \in S$ a *carrier* A_s.
2. A function $\mathcal{I}$, which assigns to each constant an *interpretation*, such that:
   (a) To each $R \in \mathcal{R}$ we assign a fundamental relation $\mathcal{I}(R)$.
   (b) To each $f \in \mathcal{F}$ we assign a fundamental function $\mathcal{I}(f)$.

   These functions are consistent with the signature.

Definition 3.1.4 (Valuation). An *valuation* in a model $\mathcal{M} = \langle \mathcal{D}, \mathcal{I} \rangle$ and a context Γ is a function $\mathcal{A}$ assigning each variable v of $LP(S, C, \sigma)$ a value $\mathcal{A}(v) \in \mathcal{D}(\Gamma(v))$.

Definition 3.1.5 (v-Variant). If $\mathcal{A}$ is an valuation, then a *v-variant* $\mathcal{A}_v$ of $\mathcal{A}$ is an valuation satisfying:

$$\mathcal{A}_v(v') = \begin{cases} \mathcal{A}(v') & \text{if } v' \neq v \\ \mathcal{A}_v(v) & \text{otherwise} \end{cases}$$

Next we define the semantics of terms in assertions. This is defined by a semantic function $\mathcal{E}_L[\![\cdot]\!]_{\mathcal{M},\mathcal{A}}$, which, given a model and a valuation, assigns to each term appearing in a formula a value in the semantic domain..

Definition 3.1.6 (Valuation of Terms). Let φ be a term of $LP(S, C, \sigma)$. Then the valuation of this term is inductively defined by:

1. For each variable v we have $\mathcal{E}_L[\![v]\!]_{\mathcal{M},\mathcal{A}} \stackrel{def}{=} \mathcal{A}(v)$.
2. For each function $f \circ (t_1, \ldots, t_n)$ we have

$$\mathcal{E}_L[\![f \circ (t_1, \ldots, t_n)]\!]_{\mathcal{M},\mathcal{A}} \stackrel{def}{=} \mathcal{I}(f)(\mathcal{E}_L[\![t_1]\!]_{\mathcal{M},\mathcal{A}}, \ldots, \mathcal{E}_L[\![t_n]\!]_{\mathcal{M},\mathcal{A}})$$

Next we define the semantics of formulae.

Definition 3.1.7 (Semantic of a Formula). If $\varphi \in LP(S, C, \sigma)$, then we define the satisfaction function $\mathcal{L}[\![\cdot]\!]_{\mathcal{M},\mathcal{A}} : LP(S, C, \sigma) \to \{true, false\}$ inductively by:

1. $\mathcal{L}[\![\mathbf{true}]\!]_{\mathcal{M},\mathcal{A}} = true$ and $\mathcal{L}[\![\mathbf{false}]\!]_{\mathcal{M},\mathcal{A}} = false$.
2. $\mathcal{L}[\![t_1 = t_2]\!]_{\mathcal{M},\mathcal{A}} = true$ if and only if $\mathcal{E}_L[\![t_1]\!]_{\mathcal{M},\mathcal{A}} = \mathcal{E}_L[\![t_2]\!]_{\mathcal{M},\mathcal{A}}$.
3. $\mathcal{L}[\![R \ni (t_1, \ldots, t_n)]\!]_{\mathcal{M},\mathcal{A}} = true$ if and only if $(\mathcal{E}_L[\![t_1]\!]_{\mathcal{M},\mathcal{A}}, \ldots, \mathcal{E}_L[\![t_n]\!]_{\mathcal{M},\mathcal{A}}) \in \mathcal{I}(R)$.
4. $\mathcal{L}[\![\neg\varphi]\!]_{\mathcal{M},\mathcal{A}} = true$ if and only if $\mathcal{L}[\![\varphi]\!]_{\mathcal{M},\mathcal{A}} = false$.
   $\mathcal{L}[\![(\varphi \wedge \psi)]\!]_{\mathcal{M},\mathcal{A}} = true$ if and only if $\mathcal{L}[\![\varphi]\!]_{\mathcal{M},\mathcal{A}} = true$ and $\mathcal{L}[\![\psi]\!]_{\mathcal{M},\mathcal{A}} = true$.
5. $\mathcal{L}[\![\exists v.\varphi]\!]_{\mathcal{M},\mathcal{A}} = true$ if and only if there exists a v-variant $\mathcal{A}_v$ of $\mathcal{A}$ such that $\mathcal{L}[\![\varphi]\!]_{\mathcal{M},\mathcal{A}_v} = true$.

We will also write $\mathcal{M}, \mathcal{A} \models \varphi$ for $\mathcal{L}[\![\varphi]\!]_{\mathcal{M},\mathcal{A}} = true$ and $\mathcal{M}, \mathcal{A} \not\models \varphi$ for $\mathcal{L}[\![\varphi]\!]_{\mathcal{M},\mathcal{A}} = false$.

3.1.2 Partial Membership Logic

The many sorted theory described in the preceding section dos not handle sub-typing. We know about two approaches to handle sub-typing: order-sorted theories [16] and membership theories [25]. We use the membership theory approach, because:

- It handles sub-typing and polymorphism in an adequate manner.
- It simplifies the partial function problem.

We define a partial order $\preceq$ on types, which will model the generalisation hierarchy of UML.

Definition 3.1.8 (Subtype Relation). Let T be a set whose elements we call *types*. A *subtype relation* is a partial order on T.

Definition 3.1.9 (Signature of Partial Membership Logic). Let S be a set of sorts, T be a set of types, and $\preceq$ be a subtype relation. We require that each sort $s \in S$ has a top-element $\top_s \in T$ with respect to $\preceq$.

Each type t of a partial membership theory is associated with a monadic relation $M_t : \top_s$, where $t \preceq \top_s$, which is interpreted as the set of all individuals of type t. Then a membership assertion is a statement $M_t \ni x$, stating that the term x evaluates to a value of type t.

The statement $t \preceq t'$ is equivalent to the axiom $\forall x. M_t \ni x \rightarrow M_{t'} \ni x$. To avoid unnecessary ambiguity we will assume that if $f : w \rightarrow k$ and $f : w' \rightarrow k'$ are operator declarations in σ with $|w| = |w'|$, then $w = w' \leftrightarrow k = k'$.

Definition 3.1.10. Given a signature $\Omega = (T, \preceq, \sigma)$, a *partial* Ω*-algebra* A assigns:

- to each $\theta \in T$ a set A_θ such that for all $\theta \preceq \theta'$ we have $A_\theta \subseteq A_{\theta'}$.
- For each $f : s_1 \cdots s_n \rightarrow s \in \sigma$ a *partial* function

$$A_f : A_{\top_{s_1}} \times \cdots \times A_{\top_{s_n}} \rightharpoonup A_{\top_s} \quad .$$

Proposition 3.1.11. *If $t \preceq t'$ we have $A_t \subseteq A_{t'}$.*

Each of the partial functions can be converted to a total function, if we introduce a replacement object, which will be the value of the function when it is undefined. For each sort of our language we introduce one such replacement object of this sort. This object has to be "far away" from the real objects in our domain; in the sense that it does not appear in the carrier of each of the types we use.

We introduce this object by requiring that the top-element of each sort does not have variables.

Definition 3.1.12 (Replacement Object). For each sort s we introduce a constant $\underline{\perp}_s$ such that for any type $t \preceq \top_s$ the axiom $\underline{\perp}_s \notin M_t$ holds. We call $\underline{\perp}_s$ the *replacement object* of sort s.

Similar to the replacement object we introduce a literal nil, which represents any empty collection. It serves as the only instance of the unit type. There exists a literal for each sort in the model.

3.1.3 Class Diagrams

We simplify the definition of a UML model of the kernel model [12]. The level of detail of the kernel model is not needed for our semantics, but each model of the kernel language can be easily mapped into our formalism.

A class diagram is a structural description of an object oriented system. It consists of a set of classes, their attributes and operations, associations and the generalisation hierarchy.[1]

[1] The generalisation hierarchy is the inverse of the inheritance hierarchy.

Definition 3.1.13 (Class). A *class* is defined by its name, a set of attributes and a set of operations. More precisely, we define:

- Let E be a set whose members we call elementary data types. The exact set used is described in Section 6.1
- Let C be a set whose members we call *class names*. We assume that the predefined data types of Chapter 6 are not in C. We write $C^+ \stackrel{def}{=} C \cup E$.
- With each class name c we associate a set A_c of typed *attribute names*. Each attribute has a type $c' \in C^+$.
- With each class name c we associate a set O_c of typed operation names.

Classes are ordered by a generalisation relation, which is described in Section 3.1.8. For sake of compatibility with the kernel language, we do not yet equate associations with attributes. The collection of class names, generalisation hierarchy and association relation defines a class diagram.

Definition 3.1.14 (Class Diagram). A *Class Diagram* is a tuple $M = (C, \preceq, A)$, where

- C is a finite set of classes, actors, and signals.
- $\preceq \subseteq C \times C$ is the transitive and reflexive closure of the generalisation relation between classes.
- $A \subseteq C \times C$ is the association relation between classes.

Remark 3.1.15. A model as defined in Definition 3.1.14 can be obtained from the model $M = (C', A, Sig, c_0, \leftrightarrow_c, \leftarrow_c, \perp_w, \leftarrow_w, \leftarrow, \leftrightarrow, <, sm)$ of the kernel model document [12, Clause 1.1.46] by:

- $C = C' \cup A \cup Sig$
- $\preceq = <^*$.
- $A = \bigcup\{\leftrightarrow_c, \leftarrow_c, \perp_w, \leftarrow_w, \leftarrow, \leftrightarrow\}$.

A class diagram induces a signature of a partial membership theory in the following way:

- Assume a special sort Any, which is the largest element of C with respect to $\preceq$.
- For each sort s assume a constant $\mathtt{nil}_s$.
- For each sort s assume a replacement object $\mathtt{undef}_s$.

Example

Using the simple class diagram of Figure 3.1 we illustrate how class diagrams are represented in a partial membership theory.

For each attribute defined in the class diagram, we generate a function of the same name. For the class diagram in Figure 3.1 we get the constant symbols

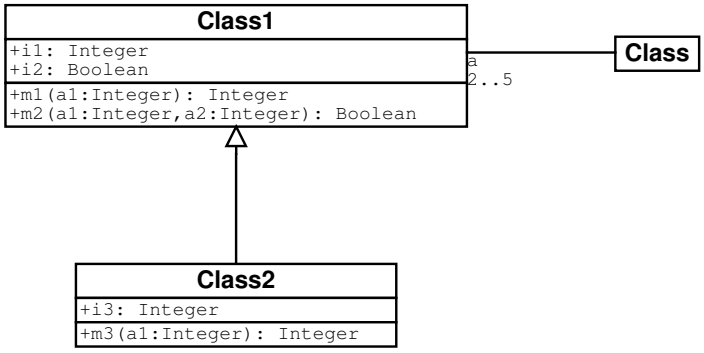

Figure 3.1: Example Class Diagram

$\{\texttt{i1}, \texttt{i2}\}$ for `Class1` and the constant symbols $\{\texttt{i1}, \texttt{i2}, \texttt{i3}\}$ for `Class2`. The symbols $\{\texttt{i1}, \texttt{i2}\}$ are inherited from `Class1`.

For each of these symbols we then create the signatures. This yields

$$\begin{aligned} \texttt{i1} &: \texttt{Class1} \to \texttt{Integer} \\ \texttt{i2} &: \texttt{Class1} \to \texttt{Boolean} \\ \texttt{i3} &: \texttt{Class2} \to \texttt{Integer} \end{aligned}$$

We have not yet considered associations. But associations are not treated differently from attributes. We have a symbol `a` with the signature

$$\texttt{a} : \texttt{Class1} \to \texttt{Set}[\texttt{Class}] \quad .$$

The multiplicity constraint on the association results in the generation of a clause of `Class1`'s class invariant:

$$2 \leq |\texttt{a}(\texttt{self})| \wedge |\texttt{a}(\texttt{self})| \leq 5 \quad .$$

3.1.4 Object Diagrams

The semantic model used for our assertions are object diagrams. These will be defined in this section. Informally an object diagram is a set of "object identifiers" $\mathbb{O}$, and set of "structural feature names" A. These names can be partitioned into "attribute names" and "association names". An attribute can be understood as an elementary value characterising an object.An association is a name, by which other objects are known. Here we do not distinguish between attributes and association names.

Each object identifier has a type. The type of an object determines which attributes and associations are defined for an object named by its identifier. The type of an object is a name of a set T. The function $\mathcal{T} : \mathbb{O} \to T$ assigns to each object its "most specific" or "minimal" type. This function corresponds to the *runtime-type information* found in many programming language implementations.

Definition 3.1.16 (Object Diagram). Let $\mathbb{O}$ be a finite set of *object identifiers*, of which each element denotes an existing object in the current state. Let C_A a finite set of *attribute names* and C_B a finite set of *association names*. $\mathcal{T}$ is a function assigning a *type name* to each object.

An *object diagram* is a pair $\mathcal{S} = (\mathbb{O}, \mathcal{I}^C, \mathcal{T})$ where $\mathcal{I}^C$ is a function which

- for each $c \in C_A$ assigns a function $\mathcal{I}^C(c) : \mathbb{O} \rightharpoonup D$, where D is the set of elementary values.
- for each $c \in C_B$ assigns a function $\mathcal{I}^C(c) : (\mathbb{O} \rightharpoonup \mathbb{O}) + (\mathbb{O} \rightharpoonup 2^{\mathbb{O}}) + (\mathbb{O} \rightharpoonup \mathbb{O}^*)$:
  - If $\mathcal{I}^C(c) : \mathbb{O} \rightharpoonup \mathbb{O}$, then c is an association of multiplicity 0 or 1.
  - If $\mathcal{I}^C(c) : \mathbb{O} \rightharpoonup 2^{\mathbb{O}}$, then c is an association of a higher multiplicity with stereotype "unordered".
  - If $\mathcal{I}^C(c) : \mathbb{O} \rightharpoonup \mathbb{O}^*$, then c is an association of a higher multiplicity with stereotype "ordered".

This definition of an object diagram can be understood as a model of a heap. It is also similar to a model introduced by Ole-Johan Dahl, which is used to handle *Pointer Aliases* [11]. The idea is to represent the object diagram as sequences of objects of the same class.

The intuition of this model is, that we consider arrays of attributes or associations, which are indexed by object identifiers, and which contain the values of the objects attributes. If the object does not have this attribute, we use the value $\perp$, which motivates the use of partial functions.

3.1.5 Navigation

Given an object diagram and a static structural description, we can now define navigation expressions. Intuitively, a navigation is the traversal of the object diagram through associations.

For simple and unordered association, we can represent the navigation operations in an relational calculus. For this, we extend the theory with a language of relational expressions.

Definition 3.1.17 (Relational Expressions). Let $LP(S, C, \sigma)$ be a theory and R be the set of all relational constants in $LP(S, C, \sigma)$. Then the language of relational expressions of $LP(S, C, \sigma)$ is:

1. Each $r \in R$ is a relational expression.
2. If r is a relational expression, then r^- is a relational expression, called the inverse of r.
3. If r is a relational expression, then $\bar{r}$ is a relational expression, called the complement of r.
4. If r and r' are relational expressions, then $(r; r')$ is a relational expression.
5. If r is a relational expression of sort ss' and t is a term of sort s, then $r[t]$ is a term of sort (set of) s'.

The formal semantics of relational expressions is expressed by a function $\mathcal{R}_L[\![\cdot]\!]_{\mathcal{M},\mathcal{A}}$.

Definition 3.1.18. The semantics of a relational expression is inductively defined by:

1. $\mathcal{R}_L[\![r]\!]_{\mathcal{M},\mathcal{A}} = \mathcal{I}(r)$ if $r \in R$.
2. $\mathcal{R}_L[\![r^-]\!]_{\mathcal{M},\mathcal{A}} = \{(b,a) : (a,b) \in \mathcal{R}_L[\![r]\!]_{\mathcal{M},\mathcal{A}}\}$.
3. $\mathcal{R}_L[\![\bar{r}]\!]_{\mathcal{M},\mathcal{A}} = \{(a,b) : (a,b) \notin \mathcal{R}_L[\![r]\!]_{\mathcal{M},\mathcal{A}}\}$.
4. $\mathcal{R}_L[\![r;r']\!]_{\mathcal{M},\mathcal{A}} = \{(a,c) : \exists b.(a,b) \in \mathcal{I}(r) \wedge (b,c) \in \mathcal{I}(r')\}$.
5. $\mathcal{E}_L[\![r(t)]\!]_{\mathcal{M},\mathcal{A}} = \{b : (\mathcal{E}_L[\![t]\!]_{\mathcal{M},\mathcal{A}}, b) \in \mathcal{R}_L[\![r]\!]_{\mathcal{M},\mathcal{A}}\}$.

Definition 3.1.19 (Navigation). Let $\mathcal{S} = (\mathbb{O}, \mathcal{I}^C, \mathcal{T})$ be an object diagram, $\mathcal{A}$ a structural description, o be an object identifier, and C_B the set of associations for $\mathcal{S}$. The a *navigation* is an expression generated by the following grammar:

$$\begin{array}{lcl} \langle\text{Navigation}\rangle & ::= & \texttt{self} \\ & | & \langle\text{AssociationName}\rangle \\ & | & \langle\text{Navigation}\rangle.\langle\text{AssociationName}\rangle \end{array}$$

We call the navigation `self` *trivial*.

We call navigations of the form ⟨AssociationName⟩ and `self`.⟨AssociationName⟩ *unqualified*.

We call any other navigation *qualified*.

The interpretation of a navigation expression is as follows:

- Associate to each association name a a relation R_a in our logic.
- Then translate the navigation into a relational expression:
  - If the navigation has the form self, then its value is self.
  - If the navigation has the form a, where a is a ⟨AssociationName⟩, then the translation is R_a[self].
  - If the navigation has the form $n.a$, where n is a navigation and a is an association name, then the translation is: $(R_n; R_a)$[self].

This semantics of navigation is flattening. The non-flattening navigation is achieved by replacing relation with function in the definitions above and use map functions as appropriate. We support both notions of navigation, but haven't decided on a default yet.

3.2 Recursive Definitions

OCL and ASO both support recursive definitions of predicates, relations, and functions. The semantics of such a definition is its *smallest fix-point*.

3.3 Type Theory

The logic on which our assertion language is based has to support sub-typing similar to object-oriented programming languages. Therefore we discuss the type theory used in this report. Essentially, we use a simplified version of the type theory presented in [15] and [34].

As discussed in Section 3.1.2 we distinguish between elementary types, object types, logical types and collection types.

Types can be "typed" by their *kinds*. We have the following language for kinds:

$$\begin{array}{rcl} \langle \text{kind} \rangle & ::= & \texttt{type} \\ & | & \texttt{type} \to \texttt{type} \end{array}$$

We have only simple types and type constructors which take only one argument.

Types of the kind *type* are called *primitive*. Types of the kind $\texttt{type} \to \texttt{type}$ are called *type constructor*.

Types in our language are terms of the following language:

$$\begin{array}{rcl} \langle \text{Type} \rangle & ::= & \texttt{BasicType} \\ & | & \langle \text{TyCon} \rangle (\langle \text{Type} \rangle) \end{array}$$

The basic types are defined in Sections 6.1. The only type constructors available in our language are defined in Section 6.3. We stress that we do not allow user-defined parameterized classes (templates) in our specification language and in UML models.

For two basic types we have the following inference rules:

$$\frac{t : \theta,\ \theta \preceq \theta'}{t : \theta'}$$

$$\frac{t : \theta(\theta''),\ \theta \preceq \theta'}{t : \theta'(\theta'')}$$

Usually, we do not allow the rule

$$\frac{t : \theta(\theta'),\ \theta' \preceq \theta''}{t : \theta(\theta'')}$$

It is, however, allowed for collection types, which are monotone type constructors. The other rules for typing and sub-typing are the usual rules.

Remark 3.3.1. In our type system it is not allowed to override $+ : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$ with $+ : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$, because it is unsafe. However, it is a feasible construct in OCL. We still allow this notation, but translate it to: $+(a, b) : \mathbb{R} \times \mathbb{R} \to \mathbb{N}$ with the pre-condition $M_{\mathsf{Integer}} \ni a \wedge M_{\mathsf{Integer}} \ni b$. This corresponds to checking the types at runtime.

3.4 Summary

The logical framework described in this chapter is used for state formulae. It supports subtyping by an axiomatisation of the inheritance relation, and exceptions by the replacement object. We sketched how an UML class diagram is

axiomatized in this logic. A temporal logic can be derived from this logic with the usual techniques.

The major differences to OCL's semantic foundations are, that recursive definitions have a smallest fix-point semantics, that we use a two-valued interpretation of formulae, and that equality is defined to be strong equality. The last difference is, that we distinguish between **nil** and **undefined**, while in the literature on OCL both concepts are often equated.

Chapter 4

Abstract Syntax

In this chapter we describe the abstract syntax of our assertion language. While we distinguish various modules of the assertion language when used to annotate class diagrams, we do not distinguish those layers in the abstract syntax of the assertion language.

The content of this chapter corresponds to a *meta-model* of ASO, except that we do not express it in UML. We use abstract BNF for our description.

The first section of this chapter explains the types of ASO. ASO types are either atomic or molecular (constructed from other types). All types are declared within UML class diagrams. Some data types are imported via a special module (see 6), while others are taken from the model of a user.

The second section of this chapter describes ASO expressions. ASO expressions are the atoms of ASO formula and ASO assertions.

The third section describes ASO assertions.

The fourth section describes how one can check whether a formula belongs to one of the different classes of assertions from the abstract syntax.

The final section summarises the content of this chapter and compares our abstract syntax to the meta-model of OCL.

4.1 Types

ASO is a typed language. Each expression has a type which is either explicitly declared or can be statically derived from a formula and its context. Evaluation of the expression yields a value of this type.

We distinguish two classes of types in our assertion language. The first class is called atomic. These types are the elementary types and the non-parameterised classes. The second class of types are the parameterised types.

We do not yet consider dependent types.

We have the following constructors for types:

$$\langle \text{type} \rangle ::= \langle \text{atomic} \rangle \mid \langle \text{tycon} \rangle \ \langle \text{type} \rangle$$

The semantics of types is explained in Section 3.3.

4.2 Expressions

The abstract syntax of expressions are defined as follows.

⟨expression⟩ ::= ⟨primary⟩
| ⟨navigation⟩
| ⟨operation-call⟩
| `pre` ⟨expression⟩
| ⟨if-then-else-expr⟩
| ⟨loop-expression⟩

⟨primary⟩ ::= ⟨literal⟩
| ⟨identifier⟩

⟨literal⟩ ::= ⟨integer-literal⟩
| ⟨boolean-literal⟩
| ⟨real-literal⟩
| ⟨string-literal⟩
| `nil`
| `undefined`

⟨navigation⟩ ::= ⟨primary⟩
| ⟨navigation⟩ `.` ⟨identifier⟩

⟨operation-call⟩ ::= ⟨navigation⟩ `.` ⟨identifier⟩ ⟨actual-parameters⟩
| ⟨navigation⟩ `->` ⟨identifier⟩ ⟨actual-parameters⟩

⟨actual-parameters⟩ ::= ϵ
| ⟨expression⟩ ⟨actual-parameters⟩

⟨if-then-else-expr⟩ ::= `if` ⟨expression⟩ ⟨expression⟩ ⟨expression⟩

⟨loop-expr⟩ ::= ⟨iterator⟩ ⟨expression⟩ ⟨identifier⟩ ⟨expression⟩

A difference to OCL is that the `pre` operator is applied to expressions in general and not to literals.

4.3 Assertions

The abstract syntax of assertions is summarised in this section. We do not distinguish between local and global assertions in the abstract syntax; this is done in the concrete syntax. Instead, we characterise the differences.

⟨assertion⟩ ::= ⟨expression⟩ = ⟨expression⟩
| ⟨expression⟩ `instanceof` ⟨type⟩
| ⟨relation⟩ ⟨actual-parameters⟩
| `not` ⟨assertion⟩
| ⟨assertion⟩ `and` ⟨assertion⟩
| ⟨assertion⟩ `or` ⟨assertion⟩
| ⟨assertion⟩ `iff` ⟨assertion⟩
| ⟨assertion⟩ `implies` ⟨assertion⟩
| ⟨assertion⟩ `xor` ⟨assertion⟩
| `exists` ⟨identifier⟩ ⟨assertion⟩
| `forall` ⟨identifier⟩ ⟨assertion⟩

4.4 Context Declarations

All constraints and assertions within OCL are declared within a context. Usually the context is assumed to be the complete system. But for a description of the internals of a component, a smaller context is needed. This is achieved by providing a context to a constraint.

The context declaration is represented by the following abstract syntax:

⟨constraints⟩ ::= ⟨constraint⟩
| ⟨constraint⟩ ⟨constraints⟩
⟨constraint⟩ ::= ⟨context⟩ ⟨assertions⟩
⟨context⟩ ::= `global`
| `component` ⟨identifier⟩
| ⟨class⟩
| `local` ⟨class⟩
| ⟨operation⟩
| `local` ⟨operation⟩
⟨assertions⟩ ::= ⟨classified-assertion⟩
| ⟨classified-assertion⟩ ⟨assertions⟩
⟨classified-assertion⟩ ::= ⟨kind⟩ ⟨name⟩ ⟨assertion⟩
⟨kind⟩ ::= `pre` | `post` | `inv`

This abstract grammar provides the means to classify the assertion into one of the context needed. We distinguish five different classes of context:

1. Expressions are not really assertions, because they may fail at run-time. Expressions are restricted to the expression fragment of ASO.

2. Local assertions have only a single object as a context in which it is evaluated. Assertions of this kind may not use navigation expressions which

navigate through any other object. Quantification has to be bounded by local expressions.

3. Inter-object assertions have a single object and its reachable environment as a context. Quantification has to be bounded by expressions. OCL is a language of this kind.

4. Component assertions have a single component instance as a context in which it is evaluated. For our purposes, a component instance is the set of objects, which comprise the component.

5. Global assertions have the complete system as a context in which it is evaluated. Because we allow hierarchical composition of components into other components, the complete system is a component itself, so there is no semantical difference between component assertions and global assertions.

4.5 Summary and Comparison to OCL

Our model of a type is simpler than the types used in OCL. While the representation of types in the UML metamodel is fine-grained in OCL, we feel that we do not need an extension of the UML meta model. All types of our assertion languages are instances of DataType or other UML meta-classes. Besides simplicity, extensions to the types of our formalism require less revision of a possible implementation.

The abstract syntax of expressions is in fact similar to OCL's meta model. A major difference is, that the pre operator may be applied to any expression. We note that OCL omits the navigation expression from the abstract syntax.

In OCL the logical connectives are defined as operations of a class Boolean. In fact, OCL does not have any notion of "built-in types." We have introduced the boolean connectives as own nodes of the syntax, but still maintain the operation view of OCL. See Section 6.1.1 for details. Also, we have added the `iff` connective for logical equivalence.

In OCL quantifiers are introduced as special iterator-expressions. We chose to introduce them as constructors in the abstract syntax of assertions.

We have extended OCL with component constraints and global constraints.

While OCL 2.0 adds concepts for "has sent a message" (using the `'^` operator) or "all sent message" (using the `^^'` operator), it does not provide means for talking about received messages in a convenient way. To do this, we introduce the `?` and `??` operators for denoting received messages.

We do not provide means to differentiate between glass-box and black-box specifications of the artifacts.

Chapter 5

Semantics of ASO

This chapter describes the formal semantics of the assertion language. The description of the semantics is based on the abstract syntax of Chapter 4. It provides the link between our constraints and the states of a system within the kernel model.

First we state our assumptions on the kernel model and define the necessary vocabulary needed to define the validity of an assertion. The semantics defined in this chapter are defined in terms of computations of the Omega kernel language, as defined in [12], and is parameterized over the precise meaning of the "properties" oclInState and stable. The meaning of these constructs has to be defined by the kernel model semantics.

5.1 Assumptions

The semantics of UML models in the kernel model are defined in terms of *symbolic transition system*. To each model M we associate a symbolic transition system $STS(M)$ which describes the structural and behavioural semantics of this model.

For this chapter, we assume a semantic function $\mathcal{P}[\![\cdot]\!]$, which assigns to each UML model a sequence of observable states. We do not define the notion of observable state here, but assume, that it is defined in [12]. Instead, we state our assumption on $\mathcal{P}[\![\cdot]\!]$. We use the predicate $stable : \mathbb{O}$ from the kernel model document in this section.

Assumption 5.1.1. We call a state *observable*, if there exists an object in the system configuration, which is in a stable. Then the semantic function $\mathcal{P}[\![\cdot]\!]$ assigns to a UML model M the maximal subsequence of stable states of the associated symbolic transition system $STS(M)$.

This is currently an assumption, because we currently do not have an agreement on what observable states are.

5.1.1 Granularity of Steps

An important decision to be made is the granularity of steps. This granularity is needed to define the notion of observable states. We propose to use three different granularities.

First, we use a coarse granularity, which is based on stable states of objects. A *macro-step* of an object is the transition from one stable state to its next stable state. This macro-step is called *run to completion step* in UML.

On a finer level of granularity we can identify an *transition step* of an object, where each step corresponds to a single transition of a state chart. This may sometimes be a run-to-completion step, but most of the time it is not.

Even a transition can be split into a finer granularity, in which each action of a transition corresponds to a step. Note that a sequence of actions may be an action, too. We do not consider this fine-grained semantics in this report, even though it is mentioned in the kernel model document.

Also, we cannot assume a *coarse global step*. The reason for this is, that generally not all objects are in a stable state, a condition we find necessary for the definition of such a step.

This leaves two levels of granularity: the run-to-completion step and the transition step. While a step is defined with respect to a single object, the state has to be a global entity, because we may navigate in assertions which have to hold in a stable state. This can only be defined by: The object from which we start navigating has to be in a stable state.

For use of the semantic definitions of OCL constraints we define these notions of granularity. Let $\mathcal{M}$ be the set of all possible kernel language models.

The micro-step semantics is the fine-grained semantic of UML models in the kernel model.

Definition 5.1.2 (Microstep Semantics). The *micro-step semantics* of a UML model $M \in \mathcal{M}$ is traces($STS(M)$).

The precise definition of traces and STS is given in Section 1.4 of [12].

5.1.2 States

OCL and ASO constraints are evaluated in a state or a pair of states, depending on their usage. Such a state is represented by an object diagram in UML. In the OMEGA kernel model a state is characterised by the system configuration, without the control information.

The state we use to evaluate a constraint does not contain control information, except the state configuration of UML state machines; the state configuration is needed to evaluate the location function. It does not refer to the pending request table or the event queue, because its contents can be derived from the communication history.

Within the kernel model a system is represented as a *symbolic transition system* (STS). The states of such a transition system is divided into a system configuration and a pending request table. For the semantics of ASO we only use the system configuration variable.

The *system configuration* of the kernel model is a function $sconf : C \to \mathbb{N} \to V \to \mathbb{V}$ mapping classes to integers to object states. We identify the set $\mathbb{O}$ of object identifiers with $C \times \mathbb{N}$. Then $sconf : \mathbb{O} \to V \to \mathbb{V}$.

In our formalism, a UML class diagram is defined by:

- A set of attributes and associations A, which are a subset of the functions of a partial membership algebra.

- The valuation function $\mathcal{A}$, which assigns to each variable occurring in a formula a value.

An interpretation of an ASO formula is now derived by

$$\mathcal{I} \downarrow A \stackrel{def}{=} \lambda v.\lambda o.sconf(o, v) \quad . \tag{5.1}$$

Therefore we focus on describing which states of a computation within the kernel model we use to evaluate an ASO (or OCL) formula.

The first semantic relation given for our work is the satisfaction relation on any state. We recall that a class diagram given in the Omega kernel language induces a partial membership theory (see Section 3.1.3).

Definition 5.1.3 (State Satisfaction). Let φ be any ASO formula and s any state of a Model M. Then $M, s \models \varphi$ if and only if s is an M algebra (see Definition 3.1.16) and $s \models \varphi$ according to Definition 3.1.7.

Based on this we define that a trace satisfies a formula if it holds for all states of the trace. Formally we will write $(\sigma, i) \models \varphi$ if the ith state of the trace σ satisfies φ.

Definition 5.1.4. Let φ be a formula. We say that a trace σ satisfies φ, written $\sigma \models \varphi$, if each state (σ, i) appearing in the computations of the corresponding symbolic transition system $STS(M)$ satisfies φ.

Definition 5.1.5. Let φ be a formula. We say that an UML model M satisfies φ, written $M \models \varphi$, if all traces of its corresponding symbolic transition system $STS(M)$ satisfies φ.

A component is essentially a set of objects C, namely the set of all objects belonging to the component. Therefore, we use the restricted state $s \downarrow C$.

5.1.3 History of Events

We assume that the runtime system records the history of events of each object at the appropriate time. Because the history of events appears as an automatic instance variable of each object in the model, We do not need to define additional semantics for histories of events here. The communication record, which we call event and which is called message in [5], is defined in Section 6.6. The data type of a history is described in Section 6.7. There you will find a definition of the operations on histories and their semantics, which can be expressed in ASO itself.

5.2 Invariants

Having defined the basic assumptions for our semantics document, we define the semantics of invariants.

5.2.1 Class Invariants

In practise it is not very useful if a class invariant has to always hold. Quite often an operation has to invalidate a class invariant in order to proceed with the computation.

For certain recursive invocations, it is also not feasible to require the establishment of a class invariant whenever the flow of control leaves the object.

A class invariant has to hold, whenever an object belonging to this class is in a stable state. This implies, that the class invariant also holds, whenever an operation is called, and when an operation is handled.

Definition 5.2.1. We say, that a class C of a UML model M satisfies its class invariant φ, if and only if for each instance c of C and each stable state of c the constraint φ holds, where the free variable self is replaced by c. More formally:

$$M, C \models \varphi \text{ iff } \forall o.\forall \sigma.\forall i.M_C \ni o \wedge \sigma \in \text{traces}(M) \rightarrow (\sigma, i) \models (\mathsf{self.stable} \rightarrow \varphi)[\mathsf{self}/o]$$

5.2.2 Component Invariants

Defining the meaning of a component invariant is more difficult. We cannot assume that there exist a situation in which all objects belonging to a component are in a stable state.

Assumption 5.2.2. Let $\mathcal{C}$ be a component. An *instance of a component* is the set of all objects comprising this component.

Definition 5.2.3. Let $\mathcal{C}$ be a component and φ be a component invariant. We say that $\mathcal{C} \models \varphi$ if and only if for all component instances and all states in which one of its objects is in a stable state the invariant holds.

5.3 Pre- and Postconditions

We currently assume partial correctness specifications for pre- and postconditions. This means, that the verifier has to establish separate proofs for these modalities [13].

Partial Correctness. An operation is partially correct with respect to its specification, if the pre-condition holds and the operation terminates, then the post-condition holds.

Success. An operation is *successful*, if whenever the pre-condition holds, no dead-lock occurs.

Convergence. An operation is converging, if the operation terminates whenever its pre-condition is satisfied.

Absence of Run-Time Errors. If the pre-condition is satisfied, prove that no run-time error can occur (for example division by zero and array index out of bounds).

Total Correctness. We call an operation totally correct with respect to its specification, if all the above modalities are satisfied.

We have not yet decided how we interpret an operational specification in ASO, and whether we introduce clauses for all those modalities, as the JML does [24]. We define the semantics of pre- and post-condition specifications as partial correctness formula.

Definition 5.3.1. Let M be a UML model, C a class of this model, c an instance of C, and o an operation with the pre-condition φ and post-condition ψ. Then $\langle\varphi, \psi\rangle$ holds for this model, written as $M, C :: o, c \models \langle\varphi, \psi\rangle$, if and only if for each computation of c in M such that c is in a stable state and is about to answer the operation call o and φ holds and the operation call terminates by returning a value, then the post condition ψ holds.

The concrete meaning of the premise is not completely resolved yet, because the kernel model language does not yet provide a consistent definition of threads. The definition is the same for primitive operations.

5.4 Expressions and Assertions

Finally, ASO constraints may appear as assertions in UML state charts. In this case, they are invariants on the state of a state chart. We do not require that the object must be in a stable state for the assertion to hold. Instead, we require that an assertion in a state machine has to hold when in stable state and after each transition. The meaning of transition is established in [12].

5.4.1 Expressions

The semantics of an ASO expression used in the expression language differs from the semantics of ASO expressions in specifications. While specifications do not show runtime-errors, expressions may.

If any sub-term of an ASO expression is evaluated to undefined, we say that the expression raises an runtime-exception. Because the kernel model language does not discuss the possibility of a run-time error we assume that the program terminates. Also, an expression may diverge. A divergent expression implies a divergent program, so we do not assign a value to the expression.

If the expression does not raise a run-time error and it is a convergence expression (which has to be proved separately), then the value of the expression in the action language is the same as its value in the specification language.

5.4.2 Assert Statement

Assertion in methods may be introduced by a special *assert* action. Then the assertion is an expression of the action language. For sake of verification we introduce a new state for each assert statement and make the assertion of the assert-statement an assertion of the state. The semantics is explained in the next section.

5.4.3 Assertions in State Diagrams

States of a state diagram may have constraints, as any other model element in UML. For each constraint φ attached to a state s in a model M its meaning is defined by

$$M \models \varphi \text{ iff } \forall\sigma.\sigma \in \text{traces}(M) \rightarrow M, \sigma \models (\mathsf{self.oclInState}(s) \rightarrow \varphi)[\mathsf{self}/o]$$

The precise semantics of oclInState is left open. We suggest, that during any transition the state chart is not in one of its states. By this the state diagram of Figure 5.1 satisfies the assertion $x \geq 0$ of state A.

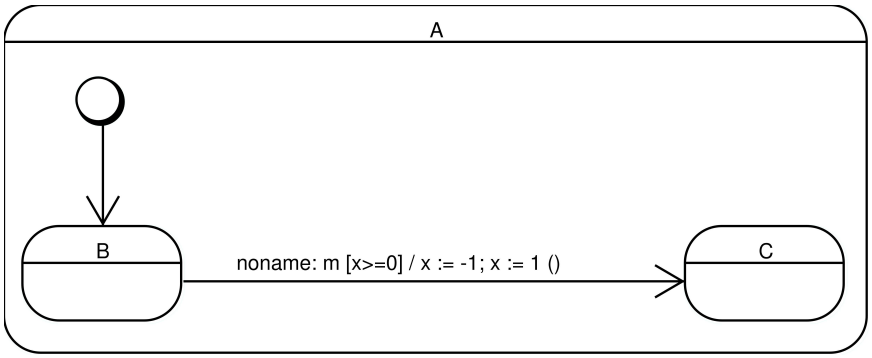

Figure 5.1: A State Machine

5.5 Summary

In this chapter we have formally defined various notions of satisfaction of OCL constraints. These definitions were related to the corresponding definitions of the kernel model. Especially, we have defined, when a constraint holds for a kernel model artifact.

Chapter 6

Predefined Data Types

In this chapter we describe the library of standard data types and define a formal semantics for these types and their operations. The data types are derived from the data types defined in UML 2.0 [2] and in OCL2.0 [5]. A summary of the data types defined in our assertion language is shown in Figure 6.1.

Data types are divided into three categories.

Elementary data types are all data types which do not refer to real object identifiers. Instances of these data types do not provide constructors, but each instance of the data type is represented as a literal.

Reference data types are all data types which refer to object identifiers. Reference data types have constructors, and they usually don't have a literal for each of their instance.

Collection data types are all data types representing collections of instances.

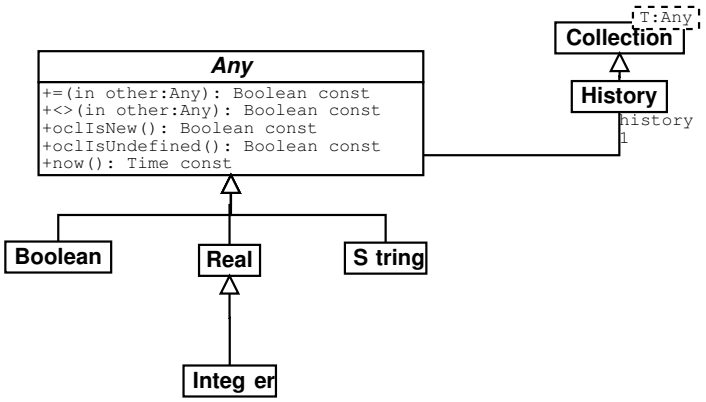

Figure 6.1: Types in ASO

6.1 Elementary Data Types

In this section we define the elementary data types which we will use. These data types will be embedded into the logic as defined in Chapter 3. Figure 6.2

summarises the elementary data types.

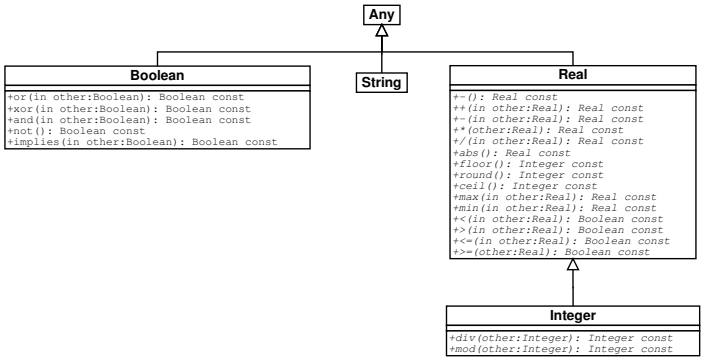

Figure 6.2: Elementary Data Types

6.1.1 Booleans

The boolean data type appearing in programs and in expressions is different from the logical truth value appearing as values of assertions. The difference is discussed in Section 2.2.6; compare also Table 2.1 and Table 2.2 for an example of the semantic differences. Therefore it is important to separate the "type of an assertion" and the type of booleans appearing in a program or UML model. The distinction is needed to avoid the semantic differences between both types, such that boolean expressions of the programming languages cannot be mistaken for assertions. Boolean expressions defined in programming languages may diverge or raise an exceptions, while our assertions do not.

The UML data type Boolean is basically the same data type (see [30], Section 2.7.2.4, p. 2-87). A boolean expression e can be converted to an assertion by a statement of the form $e = \mathsf{true}$.

Literals

The data type Boolean has two constants: true and false. Their semantics are $[\![\mathsf{true}]\!] = \mathit{true}$ and $[\![\mathsf{false}]\!] = \mathit{false}$.

Operations

not This is the negation of the programming language.

and This is the conjunction of the programming language.

or This is the disjunction of the programming language.

implies This is the implication of the programming language.

xor This is the exclusive disjunction of the programming language.

Those operations may be used method-invocation style notation and in infix notation. The semantics of this data type are defined in Table 2.2, but the semantics should be adapted to reflect the targeted programming language.

6.1.2 Real

This type represents the mathematical concept of real numbers.

Operations

$-$ The unary operator $-$ returns the negation of the real number.

$+$ The binary operator $+$ returns the sum of its arguments.

$-$ The binary operator $-$ returns the difference of its arguments.

$*$ The binary operator $*$ returns the product of its arguments.

/ The binary operator / returns the quotient of its arguments.

abs() Returns the absolute value of self.

floor() The largest integer which is less than or equal to self.

round() The integer which is closest to self. If there are two of them, then return the larger one.

ceil() The smallest integer which is greater than or equal to self.

max(other) The larger of the values self and other.

min(other) The smaller of the values self and other.

$<$ Returns true if self is less than other.

$>$ Returns true if self is greater than other.

$<=$ Returns true if self is less than or equal to other.

$>=$ Returns true if self is greater than or equal to other.

6.1.3 Integers

The data type Integer is defined with its usual constants and operations. As with Booleans, operations may be written in the algebraic way or as method invocations.

UML's Integer type is defined in Section 2.7.2.10, p. 2-89 of [30]. The carrier of this type is $\mathbb{Z}$. The following methods are defined on integers.

Operations

div The number of times other fits completely into self.

mod The result of self mod other.

All other operations are inherited from its super-type Real.

6.1.4 String

We currently do not support string as a data type. This will change in one of the next editions, when needed.

6.2 Logical Data Types

To make it easier for the specifier to write specifications, we introduce a set of logical data types: Any, Void, and Unit.

6.2.1 Any

Similar to OclAny in OCL we introduce a type *Any*, which is the common super-type of all types appearing in an UML model and the super-type of all *elementary* data types of the OCL expressions. It is not a super-type of collections and events. The type Any provides a set of query operations, which make specifying easier. The type is summarised in Figure 6.1.

Operations

The following operations are defined on Any.

= This is the strong equality operator. It returns true, if the two arguments, which are object references, are equal. If both arguments are undefined, then = returns *true*.

<> This is the negation of =.

oclIsNew() This operation can only be used in a post-condition of an operation. It returns true, if and only if the object was created during the execution of the operation.

oclIsUndefined() This operation returns true, if and only if the object this method is applied to, is the undefined object.

oclInState() Returns true if and only if the object self has got a state-chart and it is in the state named by the argument. Within OCL the state names of a state diagram are available as an enumeration type.

oclStable() Returns true if and only if the object self is in a stable state, i.e., its run to completion step has finished. You need not use this operation in an invariant, a pre- or a postcondition, because it always evaluates to true in these assertions.

history() This query operation returns the current history of this object.

now() This operation returns the current global time.

All objects have access to the global current time for sake of specification. By this, we can use a proof theory like the one presented in [20] as a basis for further development.

6.2.2 Unit

Unit is a special data type, which essentially serves the same purpose as `void` in C, C++, and Java. Unit has got one instance, which is equal to all empty collections. The instance of Unit, which we call nil, does not define operations.

6.2.3 Void

Void is the data type without any regular instances. We assume that the undefined value is also of type Void. A method or procedure declaring to return a value of type Void must not terminate. It may terminate in an *abnormal* way. If it returns, it should declare to return Unit. We assume that Void is subtype of all types appearing in a model.

Remark 6.2.1. The existence of Void is only of theoretical interest. We introduce it here, because it has its counterpart in OCL 2.0. It will usually not be used in our logic, except for the non-termination purpose.

6.3 Basic Collections

We use the same collections that were proposed for OCL 2.0 with roughly the same methods. They are the usual ones from mathematics: Set, Bag, and Sequence. As in OCL, these collections are parameterised over the type of their contents. They have Collection as their super-type. An overview of the collection types and their features is given in Figure 6.3.

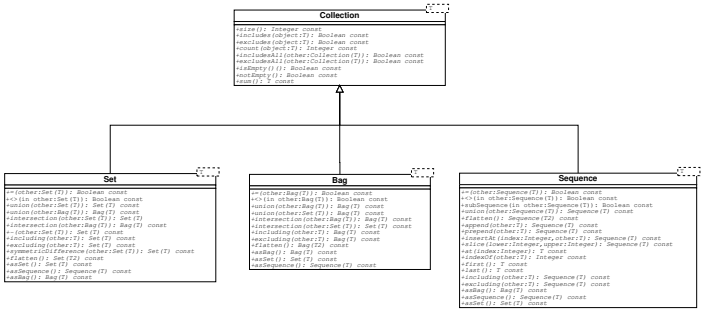

Figure 6.3: Collection Data Types

Example 6.3.1. The type Set(Integer) refers to a set of integers, while the type Set(Any) may contain almost anything.

6.3.1 Collection

We provide the same collection data types as OCL. As noted in Section 3.1.2 the object nil is representing any empty collection. This is what a programmer would expect from this literal. More importantly, nil is *not* an undefined value.

Operations

size The number of elements in the collection self.

includes True if other is an element of self, false otherwise.

excludes True if other is not an element of self, false otherwise.

count Returns the number of times the object other appears in the collection self.

includesAll Returns whether the collection self contains all elements of the collection other.

excludesAll Returns false, if the collection self contains any element of the collection other. Otherwise it returns true.

isEmpty Returns whether the collection self is empty.

notEmpty Returns whether the collection self is not empty.

sum() Returns sum of all elements of all elements of self. If the type T does not support a binary operation +, then the result is undefined. If the operation + is defined but not associative and commutative, the result is non-deterministic.

Remark 6.3.2. The operation sum() on collection is not well formaliseable. The OCL standard does not state, what happens if the criteria on the type T are *not* satisfied. If + is defined on instances of T, and it is associative and commutative, then sum() returns a determined value.

Obviously, if T does not define +, an exception will be thrown by an implementation. This is represented by having sum() return undefined.

If, however, + is defined, we have the proof obligation that + is associative and commutative. This problem is not that obvious to specify in OCL or ASO. We do not handle this situation yet, but leave the operation undefined in this case. This means that the operation may terminate normally and return some value.

6.3.2 Bag

The Bag type is used to formalise the event queue of an active object. It is the same as the mathematic bag or multi-set.

Operations

= Returns true, if and only if the bag self contains the same elements as other.

<> Returns false, if and only if the bag self contains the same elements as other.

union(other: Bag(T)) Returns the union of the Bags self and other.

union(other: Set(T) Returns the union of the Bag self and the set other.

intersection(other: Bag(T)) Returns the intersection of the Bags self and other.

intersection(other: Set(T)) Returns the intersection of the Bag self and the set other.

including Returns a bag containing all elements of self and one more element other.

excluding Returns a bag containing all elements of self without all occurrences of other.

flatten Returns the bag flattened.

asBag Returns a copy of self.

asSet Returns self converted to a set.

asSequence Returns the bag as a sequence. The order of elements in the result is randomly chosen.

6.3.3 Set

A set is an unordered collection, in which each element only appears at most once. We use sets in our formalism to express reachability properties.

Operations

The type set defines the following operations:

union(other: Set(T)) Returns the union of self and other.

union(other: Bag(T)) Returns the union of self and other.

= Returns true, if self is equal to other. Two sets are equal, if they contain the same elements.

<> Returns true, if self is not equal to other.

intersection(other: Set(T)) Returns the intersection of self and other.

intersection(other: Bag(T)) Returns the intersection of self and other.

− Returns the difference of self and other.

including Returns a set containing all elements of self and other.

excluding Returns a set containing all elements of self without other.

symmetricDifference Returns the symmetric of self and other.

flatten Flattens self, is applicable.

asBag Returns the elements of self in a bag.

asSequence Returns the elements of self in a sequence. The elements are in a random order.

asSet Returns a copy of self.

The subset relation is called includesAll() in OCL. Cardinality of a set is computed by size()

Changes to OCL 2.0

The following changes were made to the OCL 2.0 data type Set.

- An operation <> has been added.

6.3.4 Sequence

A sequence is a collection, of which the elements are ordered. An element may be part of a sequence more than once.

Operations

The following operations are defined on Sequence, in addition to the ones inherited from Collection.

= Returns true if and only if the two sequences contain the same elements in the same order.

<> Returns true if and only if the two sequences are different.

subSequence Returns true if and only if the sequence other is a subsequence of self.

union Returns all elements of self followed by all elements of other.

flatten Flattens the sequence.

append Returns the sequence consisting of self followed by the element other.

prepend Returns the sequence starting with other followed by all elements of self.

insertAt Returns the sequence self with other inserted at position index.

slice Returns the subsequence of self starting at lower and ending at upper.

at Returns the element at position index.

indexOf Returns the index of the first occurrence of other in self.

first Returns the first element of self.

last Returns the last element of self.

including This is a synonym for append.

excluding Returns the sequence containing all elements of self without all occurrences of other.

asBag Returns a bag containing all elements of self.

asSequence Returns a copy of self.

asSet Returns a set containing all elements of self.

Changes to OCL 2.0

The Sequence type of OCL 2.0 was changed in the following ways to be more usable:

- An operation <> was added, to allow shorter specifications.
- The operation OCL operation subSequence was renamed to slice.
- An operation subSequence was added. This is a relation which determines whether self is a subsequence of other.

6.4 Iterating Collections

One of the most powerful language constructs in OCL is the *iterate* expression. In this section we introduce our versions of this expression. Instead of one standard iterator, we introduce two: *map* and *fold*. For sets and bags the fold construct of our assertion language corresponds to the OCL iterate expression. For sequences we distinguish the direction of the folding operation.

The general format of the iterate expression is:

```
collection->iterate(elem: Type; acc: Type = <expression> |
                    <expression-with-elem-and-acc>)
```

The variable *elem* is the iterator variable.

The variable *acc* is the accumulator variable. The value of the iterate expression is assigned after ...

An *iterate* expression is evaluated by first assigning the value of *expression* to the accumulator variable, then we let *elem* iterate over each element of *collection*. For each value of *elem* the iterator expression is evaluated and its value is assigned to *acc*. If all elements have been handled, the value of *acc* is returned.

The order in which the elements are handled is not defined for Set and Bag, for Sequence it is the order of the elements within the sequence.

Formally, we declare the semantics of the *iterate* expression by defining a recursive function with the signature $S^* \to T \to (S \to T \to T) \to T$ as:

$$iterate(S, a, e) = \begin{cases} a & \text{if } S = \langle\rangle \\ iterate(S', e(a, s), e) & \text{if } S = \langle s \rangle \cdot S' \end{cases} . \tag{6.1}$$

Because we have coercion functions from any collection types to sequences, and all other OCL iterators can be expressed using the *iterate* function, this definition is all we need to express the semantics of the iterate functions.

Nested iteration expressions are defined with the help of the following iterator-expression:

$$iterate_2(S, a, e) \stackrel{def}{=} iterate(S, a, \lambda x.iterate(S, a, e(x))) \tag{6.2}$$

We can generalise this to any number of iterator variables n by inductively defining:

$$iterate_n(S, a, e) \stackrel{def}{=} iterate(S, a, \lambda x.iterate_{n-1}(S, a, e(x))) \tag{6.3}$$

6.4.1 Predefined Iterators on Sequences

The standard iterator expressions for sequences are:

select The subsequence of the source sequence for which body is true.

$$select(S, b) \stackrel{def}{=} iterate(S, \langle\rangle, \lambda a.\lambda s.\text{if } b(s) \text{ then } a \cdot \langle s \rangle \text{ else } a)$$

reject The subsequence of the source sequence for which body is false.

$$reject(S, b) \stackrel{def}{=} iterate(S, \langle\rangle, \lambda a.\lambda s.\text{if } b(s) \text{ then } a \text{ else } a \cdot \langle s \rangle)$$

collectNested The sequence of elements which result from applying body to every member of the source sequence. This may also be called *map*.

$$collectNested(S, b) \stackrel{def}{=} iterate(S, \langle\rangle, \lambda d.\lambda a.a \cdot \langle b(d) \rangle)$$

6.4.2 Predefined Iterators on Bags

The standard iterator expressions for bags are derived from the iterator expressions for sequences. We just summarise their definitions:

$$\begin{aligned} select_{Bag}(B, b) &\stackrel{def}{=} asBag(select(asSequence(B), b)) \\ reject_{Bag}(B, b) &\stackrel{def}{=} asBag(reject(asSequence(B), b)) \\ collectNested_{Bag}(B, b) &\stackrel{def}{=} asBag(collectNested(asSequence(B), b)) \end{aligned}$$

6.4.3 Predefined Iterators on Sets

The standard iterator expressions for bags are derived from the iterator expressions for sequences. We just summarise their definitions:

$$\begin{aligned} select_{Set}(B, b) &\stackrel{def}{=} asSet(select(asSequence(B), b)) \\ reject_{Set}(B, b) &\stackrel{def}{=} asSet(reject(asSequence(B), b)) \\ collectNested_{Set}(B, b) &\stackrel{def}{=} asSet(collectNested(asSequence(B), b)) \end{aligned}$$

6.4.4 Predefined Iterators on Collections

Given this definition, we recall the definitions of the iterators from OCL:

exists Bounded existential quantification over collection elements.

$$\begin{aligned} exists_1(S, b) &\stackrel{def}{=} iterate(S, false, \lambda a.\lambda x.a \vee b(x)) \\ exists_n(S, b) &\stackrel{def}{=} exists(S, false, \lambda x.exists_{n-1}(S, b(x))) \end{aligned}$$

A similar construction is defined for other nested iterators.

forall Bounded universal quantification over collection elements.

$$forall(S, b) \stackrel{def}{=} iterate(S, true, \lambda a.\lambda x.a \wedge b(x))$$

isUnique Results in true if and only if the *body* evaluates to a different value for each element in the collection.

$$isUnique(S, b) = forall_2(collect(S, b), \lambda x.\lambda y.x <> y)$$

sortedBy Results in a Sequence containing the same elements as the source collection, but sorted by *body*. The expression *body* should map into a domain with the $<$ operation defined, which defines a linear order. *sortedBy* must not define more than one iterator variable.

any Returns any element in the source collection for which *body* evaluates to true.

$$any(S, b) \stackrel{def}{=} first(asSequence(select(S, b)))$$

one Results in true if these is exactly one element in the source collection for which body is true.

$$one(S, b) \stackrel{def}{=} |select(S, b)| = 1$$

collect The collection of elements which result from applying *body* to any element of the source collection.

$$collect(S, b) \stackrel{def}{=} flatten(collectNested(S, b))$$

6.5 Time

In this section we present the data structures necessary for handling real time constraints. Figure 6.4 summarises our time extension. We introduce two concepts: Time and Duration.

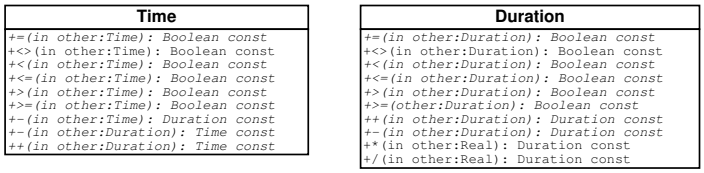

Figure 6.4: Time Data Types

6.5.1 Time

Instances of type Time refer to the values provided by a global clock. For sake of simplicity we interpret the values of Time as real numbers. The only way to produce instances of this type is by invoking the operation now defined in Any (see Section 6.2.1).

Operations

The data type Time defines the following operations:

= Returns true if and only if the instance self is equal to other.

<> Returns false if and only if the instance self is equal to other.

< Returns true if and only if the instance self is before to other.

<= Returns true if and only if the instance self is before or equal to other.

> Returns true if and only if the instance self is after other.

>= Returns true if and only if the instance self is after or equal to other.

− Returns the duration between self and other.

− Returns the time other time units (which is a Duration) before self.

+ Returns the time other time units (which is a Duration) after self.

6.5.2 Duration

The data type Time defines the following operations:

= Returns true if and only if the instance self is equal to other.

<> Returns false if and only if the instance self is equal to other.

< Returns true if and only if the instance self is less than other.

<= Returns true if and only if the instance self is less than or equal to other.

> Returns true if and only if the instance self is greater than other.

>= Returns true if and only if the instance self is greater than or equal to other.

+ Returns the sum of both durations.

− Returns the difference between the two durations.

∗ Returns the product of self and other.

/ Returns the quotient of self and other.

6.6 Messages and Events

Events drive the computation of UML models. An *event* is a specification of a type of observable occurrence. The occurrence that generates an event instance is assumed to take place at an instant in time with no duration. The UML standard distinguishes four kinds of events: signal events, call events, time events and change events. Of these, we only consider Signal and Call events, which describe the communication between objects.

A *call event* models the occurrence of a operation call between two objects.

A *change event* refers to the evaluation of boolean expressions within a state machine. Whenever such an expression is evaluated to true, a change event is generated. These are messages send from an object to itself, and therefore not observable by other objects.

A *signal event* models the asynchronous communication between two objects.

A *time event* models the expiration of a specific deadline. Our model is untimed, therefore we do not consider them in this report.

A *message* is a particular instance of a call or signal event. A *history* a sequence of messages and waiting events. We introduce data types for these concepts.

The event queue for a state machine is associated with an instance of this classifier.

6.6.1 Communication Record

In this section we explain the basic structure of a *communication record*. The communication record stores the information needed for our histories. In addition to the different event types described above, an event also has got a life-cycle. The events life cycle is:

completed We call an event *completed*, if the reaction to the event is completed after the it was dispatched.

deferred We call an event *deferred*, if it is going to be dispatched, but the receiving object is not yet ready for receiving this event. The event remains in the event queue, and another event may be chosen.

delivered We call an event *delivered*, if an action occurs such that it is delivered to one or more targets; it then is part of the targets event queues.

discarded We call an event *discarded*, if it is dequeued from the event queue, but no transition can be triggered by this event, and it cannot be deferred.

dispatched We call an event *dispatched*, if it is dequeued from an event queue and handed to an object for execution.

In [18] the life cycle of a message is simplified to *send, receive*, and *consume*. We follow this simplified approach, and introduce discarded and dispatched as derived properties.

The communication record has the following members:

sender The object which send the message.

receiver The object which received the message.

kind The kind of the event. This is either *signal* or *operation*. As noted above, we do not support *condition* and *time* events.

name The name of the event.

parameters A list of parameter values. The parameter values have to be consistent with the signal or operation declaration.

state One of the values from the life cycle. This is one of *sent, received*, or *consumed* for signals or *invoke, receive, consume, return, receive-return*, or *consume-return* for operation call messages. The state may have other fields, as defined in [18].

result If the modelled element is a operation call and the record is a return record, this member contains the return value.

time This member records the time, at which the event occurred.

The communication record defines a set of operations which makes using them easier. These extensions are described in the following sections.

Example 6.6.1. The OCL 2.0 operator `o^^msg(e)` is equivalent to the ASO expression

```
history->select(m | m.sent() and m.receiver = o and
                    m.name = 'msg' and m.parameters = e)
```

6.6.2 Signal Events

Signal events do not cause any synchronisation between the sending object and the receiving object. We record signal events with one communication record on the sender's history and two communication records on the receiver's history.

send This record models that a signal was send from the sender to the receiver. It appears only in the senders history.

receive This record models that a signal was received by the receiver and is put into the event queue. It appears only in the receivers history.

consume corresponds to the time when the event is handled by the object. It appears only in the receivers history.

For interface specifications the consume event is generally not useful. The main reason is that the consume event refers to a particular implementation, because it also stores the transition which was taken when the event is consumed. This is useful for white-box specifications, but not applicable for black-box specification. We also introduce two derived events:

accept This record models that the signal was received by the object and was dispatched. It essentially corresponds to the statement "there exists a transition such that the event is consume transition."

reject This record models that the signal was received by the object and that it was rejected. It essentially corresponds to the statement "there exists no transition such that the event is consume transition."

6.6.3 Call Event

Call events modell the process of invoking an operation between two objects. We model the operation call by two signals sent between the two participating objects: one signal is used to invoke the operation, the other is used to send a return value back and to report completion of the operation call. The communication records generated for a call event are

invoke The invoke record records the fact that the caller object c calls the operation or method o of the called object d with actual parameter values $\vec{p}$ at time t. This event appears in both histories. Because a synchronisation happens, both records carry the same time stamp.

receive This event records that an operation call has been received and is waiting to be handled.

consume This event records that an operation call message is accepted by the object.

return The return record models, that the called object d returns from an operation call of the operation or method o from object c with actual parameter values $\vec{p}$ at time t with value v.

receive-return This event records that the return value of an operation call has been received and is waiting to be handled.

consume-return This event records that the return value of an operation call is consumed and the blocked object may proceed with its computations.

Similar to signals, the consume records may only be used in white-box specifications, because they store the corresponding transition in the state chart. This information is not available for black-box specification. Therefore, we define the derived records *accept, reject, accept-return,* and *reject-return.*

6.6.4 Object Creation

Object creation is part of the history. It is recorded by a variant of the call event. In case of object creation, the receiver of the constructor call is considered to be a virtual object having the name of the class of which we create the object. The result is the identifier of the newly created object.

Example 6.6.2. If we encounter the statement $o :=$ new Class(e_0) in a Java program, we have the following call event: $\langle \mathsf{self}, \mathsf{Class}, \mathit{call}, \mathsf{Class}, \langle e_0 \rangle, s, o, t \rangle$, where self is the value of the sender, Class is the virtual object we send the message to and the name of the constructor method, *call* is the event call, $\langle e_0 \rangle$ is the list of parameter values sent to the constructor, and o is the identifier of the newly created object.

Remark 6.6.3. We have chosen to model object creation in this way, because it allows more flexibility in the conventions used for constructors. UML 2.0 attaches the stereotype "constructor" to a operation in order to make it a constructor. The same convention is used in the programming language Eiffel [26]. This scheme allows for two different constructors with the same signature, which you cannot have in C++ [37] or Java [17].

6.7 Histories

We assume a mechanism which updates the history of events whenever necessary.

To reason compositionally about object structures we use communication histories. Two kinds of histories are maintained: we maintain a global history, which records the events send between all objects, and each object maintains its own local history. Histories are of the type `Sequence[Event]`. Generally speaking, a *communication history* is a sequence of events (formally: `History` $\stackrel{def}{=}$ `Sequence`[`Event`]).

We do not discuss the mechanism which maintains the histories within this document. Instead, we assume that a modified version of the interpreter described in the kernel model will update the history when necessary.

We define a set of operations on histories which allow us to write down *past-time temporal logic* formulae, albeit in an object-oriented fashion. on communication events. With these predicates on histories, the user should have an expressive tool for specifying the order of events.

We use the convention that each predicate combined with and applied to the predefined temporal operators take a history as first argument.

sofar(expr) This predicate states that p holds for every prefix of the history.

```
sofar(expr) = forall(i | history()->slice(1,i) = expr )
```

once(expr) This predicate states that p holds for one prefix of the history h.

```
once(expr) = exists(i | history()->slice(1,i) = expr)
```

since(e,f) This predicate states that q holds for one prefix of the history and since then p holds for every "later" prefix.

```
exists(i | history()->slice(1,i) = e and
           forall(j | j > i implies history()->slice(1,i) = f) )
```

backto(e,f) This predicate is a weak version of since.

```
backto(e,f) = sofar(e) or since(e,f)
```

previously(e) This predicate states that p holds for the prefix of h which is one event smaller.

```
previously(e) = (slice(1, history->size() - 1) = e)
```

before(e) This is a weak version of `Previously`.

```
history()->size()=0 or previously(e)
```

Example 6.7.1. That p causes q can be expressed by

$$\texttt{SoFar}(\texttt{history}, \lambda h.q(h) \rightarrow \texttt{Once}(h, p)) \quad .$$

This expands to

$$\forall i : \mathsf{Integer}.(i < |\texttt{history}|) \rightarrow (q(\mathit{slice}(\texttt{history}, 0, i) \rightarrow \texttt{Once}(\mathit{slice}(\texttt{history}, 0, i), p))) \quad .$$

This can be further expanded to:

$$\forall i : \mathsf{Integer}.(i < |\texttt{history}|) \rightarrow (q(\mathit{slice}(\texttt{history}, 0, i) \rightarrow \exists j.(j < |\texttt{history}|) \wedge p(\mathit{slice}(\texttt{history}, 0, j)))) \quad .$$

6.8 Summary

In this chapter we have defined our standard library of data types. Most data types are identical or extensions to the data types presented in OCL 2.0, but some data types have been changed in order to define a consistent set of operator names.

One lack of UML and OCL was identified during the definition of the standard library. UML and OCL do not have a notion of function. We introduced the concept of a higher order function in order to have a clean syntax and semantics for our iterators. On the other hand, the higher order functions as used in this chapter can be easily implemented in C++.

Chapter 7

Annotating UML Diagrams

In this chapter we will describe the basic pragmatics of annotating UML diagrams. It is intended to be a short tutorial on using ASO in the development progress. We explain which kind of development method we intend to support with our notation.

To make the kind of development method supported by our language extensions more apparent we have split ASO into five dialects, which distinguish their use in a diagram.

At the lowest level, ASO may be used as an expression language for initialising attributes of classes and for use as guards in life sequence charts and state charts. For this level the language should simple be efficiently computable. We disallow quantification and the OCL collection types on this level.

On the next higher level, ASO is a language for the local specification of Objects.

On the third level, ASO may describe the environment of an object. The second and third level are very similar to OCL.

On the fourth level, ASO is used for the specification of components. This level is similar to the intra-object specification level, but raised to components.

The fifth level is a inter-component specification language, which may be used to write further constraints on how components may be linked in a specific application.

As an example in this section we use a simple database supporting some dynamic set operations. These are implemented by a B-Tree data structure. B-Trees were introduced by Bayer and McCreight [4], their implementation is discussed by Knuth [22].

The example is described by the class diagram in Figure 7.1. We have defined an Interface Database, which specifies the operations insert(), contains(),

4	System specification
3	Component specification
2	Inter-object specification
1	Intra-object specification
0	Expression language

Table 7.1: Levels of Specifications

delete(), and min(). We assume that the data base collects entries of type T, which is a subclass of Comparable. The class Comparable designates an operation defining a linear order on its instances. For sake of simplicity, our entries are just Integers.

The Database interface is implemented by a BTree class. If we consider the implementation as a component, then its port would be the root node of the tree.

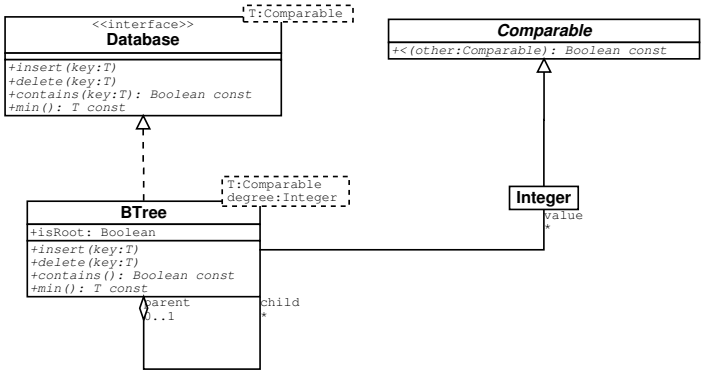

Figure 7.1: Database and B-Tree Example

7.1 The Expression Language Fragment

ASO expressions may be used as general expressions in behavioural diagrams. They may be used to describe constraints on messages in life sequence charts. They may also appear as guards in transitions. Finally they may also be used as initialiser expressions of program fragments.

ASO as expression language disallows the following constructs:

- The OCL collection types may not be used in expressions. Instead, the user must use program types. Of course, the user is free to use those types and its methods, if he defines them himself in the program.
- No quantification is allowed.

Additionally, if a method is to be invoked in an expression, the user may only do so, when:

- The operation is side-effect free.
- The user has proven, that it always terminates regularly.

Example

Consider the state machine of Figure 7.2. The guards of a transition in this figure are OCL expressions enclosed in square brackets, for example $[x >= 0]$.

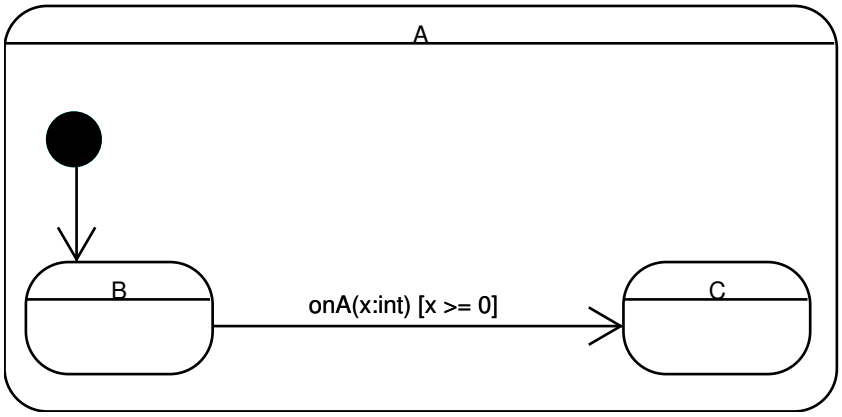

Figure 7.2: A State Machine

7.2 Intra-Object Specification

Object Specifications are specifications which require an object and its environment as a context. These specifications are of the same kind as OCL specifications. Object specifications can be used in class diagrams, life sequence charts, and state machines.

An *intra-object specification* is a specification local to an object. Such specifications are used to constrain the instance variables of an object or to create new subtypes. Their most important feature is, that such a specification is not part of an objects interface. A client of an object does not need any knowledge of these constraints.

7.2.1 Class Diagrams

The annotation of class diagrams should be done on two levels with two views. We have two views on classes. The first view is a white-box view, where all internal features of the class can be constraint. It is also useful to give black-box specifications, where only the publicly visible parts of the class may appear in a specification. This kind of specifications are highly useful for clients of a class, because they do not see the internal details of the class they want to use.

The first level of specification reasons about the local behaviour of an object should be specified using a *local* assertion language. In this local assertion language the user may not use any navigation expressions. He may also not use any operations, which make use of navigation expressions. The reason for such a specification is, because we do not want to have the specification of a class to depend too much on its context. Otherwise placing the object into another context will require a revision of its specification.

Example 7.2.1. The class diagram in Figure 7.1 shows the definition of the B-Tree data type. This data type is constraint by the following invariants:

```
context BTree
local inv: root implies parent = nil
local inv: not root implies number >= degree
```

```
local inv: number <= 2 * degree
local inv: n = value->size() and
           (child->size() = 0 or child->size() = value->size() + 1)
local inv: child->forall(i | i >= 2 and i <= number implies
                         value->at(i) > value->at(i - 1) )
```

The first invariant states that all nodes which are not the root node have at least `degree` entries. The second invariant states that any nodes has at most `2 * degree` entries. The third invariant states that the values are sorted in increasing order. All these invariants are actually not part of the interface, because the entities these invariants refer to are private. For this reason the constraints are stereotyped "local".

A diagrammatic representation of this constraint is shown in Figure 7.3.

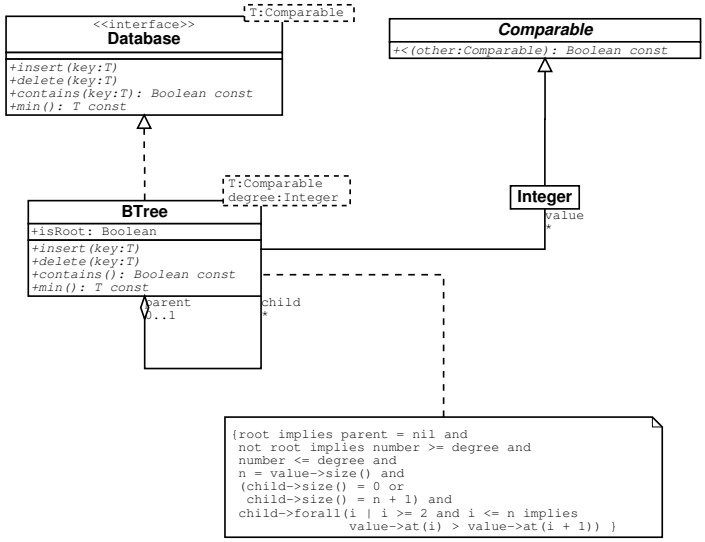

Figure 7.3: Annotated B-Tree Class Diagram

7.3 Inter-Object Specification

Most intra-object specifications are local. They usually describe how the values of a classifier are related. But objects do not live alone. They interact with other objects. This has to be specified in the inter-object specification language.

In our database example, the objects of a B-Tree collaborate to achieve a common goal. Each leaf is responsible for storing some keys, and relies on a certain global layout to find any other key. The database itself is build from many smaller objects, which collaborate in implementing the specification. The way these objects collaborate is usually specified in an inter-object specification language.

In the inter-object relation we declare a context for an object we start with, an allow navigation and bounded quantification. This relaxes the condition on intra-object specifications where we were not allowed to navigate the object structure.

Example 7.3.1. In the B-Tree the position of the smallest element is fixed. We can specify the value by defining a simple look-up function. This can be done by the following constraint:

```
context BTree::min()
pre: value->size() > 0
post: RESULT = if child->size() > 0
               then child->at(1).min()
               else value->at(1)
               endif
```

The pre-condition states that we have to have at least one value stored in the node, while the post-condition tells us how we can look up the smallest value in the B-Tree.

The problem with the specification we gave for `BTree::min()` is, that it makes use of the internal structure of a BTree. It is not suited as an interface specification. In an interface specification we want to abstract from the internal layout of an object.

On the second level we specify how an object collaborates with its environment to achieve a certain goal. We call this level of specification language the *inter-object* level. This level of specification has two purposes:

- We can define constraints of how the objects of an aggregate collaborate. This kind of specification is necessary for the specification of the internal behaviour of an active object.
- We can give further constraints on how objects link in a certain application. This kind of specification is a specialisation of the first level specification. These specifications are called component invariants in [3].

Class diagrams can be annotated in a separate document, where each context has to be explicitly declared. The annotations may also be part of the class diagram, using the constraint element of UML. In this case the user shall choose ASO instead of OCL as a specification language. In both cases, the user is free to write his constraints in either standard OCL, or in our extension ASO.

Using our terminology, OCL is a white-box inter-object specification language.

Example 7.3.2. For the B-Tree example we have the following additional invariants:

```
context BTree
inv: node->forall(i | 0 <= i and i <= number implies
                  child->at(i).min() > value->at(i) and
                  child->at(i).max() < value->at(i + 1))
```

These invariants are not local anymore, but constrain how any node in a B-Tree is related to its environment.

The same specification can also be written as a black-box specification

Example 7.3.3. We can consider the root node of a BTree as the port to a Set component. For this component we want to specify the interface behaviour of the min() operation:

```
context BTree::min()
interface def: oper contents(s, i) =
    if i < history()->size()
    then if history()->at(i).name = 'insert'
         then contents(i + 1,
              s->including(history->parameters->at(1)))
         else if history->at(i).name = 'delete'
         then contents(i + 1,
              s->excluding(history->parameters->at(1)))
         else contents(i + 1, s)
    else nil
interface pre: root and history()->select(m | m.name = 'insert'
          and m.accept)s->size() >
     history()->select(m | m.name = 'delete' and
          m.accept)->size()
interface post: contents(1, nil)->forall(i | i >= RESULT)
```

The pre-condition states that we have received more insert() operation calls than delete() operation calls. The post condition describes the smallest value inserted which was not deleted yet.

Example 7.3.4. We can use the same specification for the interface of Set. In this case the stereotype interface is implicit.

Another example illustrating the usefulness of splitting accept and reject states from received states can also be specified using histories.

Example 7.3.5. To specify that a call of the insert operation will never be rejected, write:

```
context BTree
inv: history->forall(m | m.name = 'insert' implies not m.reject)
```

Example 7.3.6. To specify that a message will be accepted within a certain time bound, write:

```
inv: let h = history->select(m | m.name = 'insert')
     in Sequence{1..(h->size())}->forall(i |
         Sequence{i..(h->size())}->forall(j | h->at(i).sender =
              h->at(j).sender implies
              h->at(j).time - h->at(i).time <= bound))
```

7.3.1 Life Sequence Charts

ASO formulae may appear in life sequence charts as general expressions, conditions, and as guards. General object specifications may only be used in activation conditions. ASO expressions may be used as conditions.

7.3.2 State Machines

Intra-Object and Inter-Object specifications can be used to constrain UML state machines. The class Any defines a *location predicate*: Using the oclIn-State(stateName) function, one can define a constraint on the states of a state machine. This function returns true, if and only if a state machine defined for

the class is in the state stateName in the context of a corresponding object. Invariants on a state machine can be specified by a predicate

$$\mathsf{inState}(s_0) \rightarrow assertion_0$$

The draw-back of this kind of specification is, that the constraint only has to hold, when the object is in a stable state, as described in Section 5.2. Therefore we suggest using the location predicate only in pre- and post-conditions of class diagrams.

The second way of annotating state machines is by adding constraints on states. Such a constraint must hold, whenever a transition enters or leaves the state, as described in Section 5.4.

7.4 Intra-Component Specification

The next two levels of specification are component specifications. As with objects and classes, we have again two levels with two views. Because we can view a component to be an object used in a codified way, the intention of this level of specification is the same as in the preceding section.

However, components are unspecified entities of OCL. ASO provides an extension to OCL in that it allows to refer to the name of a component. Constraints of a component are only valid for all objects which comprise a component. (The specification of a class should make the distinction of what we consider an internal object or external object).

Intra-component specifications are invariants on the object diagram constituting a component. In our B-Tree example, an inter-component specification states that the smallest element is in a special node of the object diagram describing the B-Tree.

Example 7.4.1. The node holding the smallest element may be specified by:

```
context 'Database'
inv: BTree.allInstances()->select(isRoot)->forall( r |
     let min_node(x) = if x.child->size() > 0
                       then min_node(x.child->at(1))
                       else x
                       endif
     in min_node(r).value->at(1) = r.min() )
```

Notice that the context 'Database' restricts the value of `BTree.allInstances()` to all instances of BTree within that component.

7.5 Inter-Component Specification

For the inter-component specification we consider a component to be a black box with a set of ports which enable us to communicate with a component. On this level, we have to enable a specifier to describe, how a message sent to a port of a component affects the behaviour of the complete component in a way, that does not refer to the internal structure of the component.

On the other hand one has to explicitly state which ports are related and how the behaviour on one port affects the behaviour of other ports of the same

component. To make this easy, we have a component history, in which all events on all ports are recorded. The history of a port object is a projection of the component's history and the external objects. Thus we hide all communication sent into the component from a port object.

To make this scheme work, a component instance needs to have a unique name, to which we associate its ports. The context of a inter-component specification is the complete system. You may not refer to any object which does not represent a port associated to a component.

Example 7.5.1. An inter-component specification may be the following follows:

```
context 'System'
inv: Any.allInstances()->forall(o | o->history()->select( m |
               m.received() and m.accept and
               m.parameters()->exists(d |
                    d.oclIsType(Database))
               ->asSet()->size() <= 1 )
```

This specification means that all components have received at most one reference to any "Database" interface.

7.6 Summary

This chapter has presented how one annotates diagrams of a model description. Most of the methods proposed here are standard for UML, and should already be available in most tools. We have given formal semantics to these annotations by referring to preceding chapters.

We may have extended OCL with a notion of visibility of constraints, as it is done in JML, for example. We feel that our way of organising constraints is better suited for documentation, because it distinguishes between how a specification is going to be used.

Chapter 8

Summary and Future Work

In this report we have described the syntax and semantics of our extension of OCL for architectural specifications.

8.1 Summary

With this report we have defined a formal semantics for an extended version of OCL. We have started from the ongoing work on OCL 2.0 [5], and added extensions to specify components, as well as support for the black-box and white-box specification of objects and components.

We have been able to remain compatible with the syntax of OCL 1.4 and OCL 2.0. Even though the semantics of our assertion language is different from OCL 1.4 and OCL 2.0, there is not too much work in translating OCL expressions into ASO expressions. This process will be automated by one of our tools.

The most important changes are, that recursive functions and predicates have a least fix-point semantics, and that our assertion language uses a two-valued semantics. The only user visible change is, that in OCL formulae most expressions e of type Boolean have to be replaced by $e = \mathit{true}$.

8.2 Future Work

In the future, we want to extend our assertion language to express liveness properties and improve its syntax and semantics to address usability and real-time.

The grammar defined in Appendix A was already implemented in a prototype parser. The parser will be extended with a type-checker and a compatibility checker. The compatibility checker will identify all language constructs of OCL 2.0 which have a different meaning in ASO, and propose corrections. This part can also be used to automate the translation from OCL 2.0 to ASO.

We will extend this prototype to generate verification conditions of models in PVS. To achieve this, the formal model described in Chapter 3nd the data types defined in Chapter 6ill be implemented in PVS.

We intend to study how we can use our assertion language for the specification and verification of software components without refering to the implementation of those components.

Appendix A

Interchange Format and Concrete Syntax

This appendix defines the concrete syntax of OCL. The syntax is based on the syntax of OCL 2.0 [5], but we have extended it and corrected some mistakes. This chapter is derived from the input file of an automatic parser generator (yacc). This grammar is able to parse all constraints from [39].

The grammar in [5] is different from the grammar presented here. Our grammar was developed for tool implementation and is nearly LALR(1). The grammar in [5] is currently incomplete and was designed for a context-sensitive predictive parser.

The grammar of this chapter with the content of Section A.6 are part of the tool exchange [28].

A.1 Keywords and Literals

We use the same keywords as OCL 2.0 and we have added a two new keywords `interface` and `local` for local assertions and assertional specifications.

`and`	`attr`	`context`	`def`	`else`
`endif`	`endpackage`	`false`	`if`	`implies`
`in`	`interface`	`inv`	`let`	`local`
`nil`	`not`	`oper`	`or`	`package`
`post`	`pre`	`then`	`true`	`undefined`
`xor`				

Table A.1: Reserved Keywords

Among these keywords we have the literals `true` and `false` which represent the truth values.

boolean-literal:
`"true"` | `"false"`

An integer literal is a positive number in decimal representation. The smallest and largest representable integer value is defined by the implementation.

integer-literal:
    [1-9][0-9]*

A real literal is a positive number in decimal representation. The exact representation of real literals is implementation defined.

real-literal:
    [0-9][0-9]*.[1-9][0-9]*

A string literal is any sequence delimited by ' characters. Special characters may be escaped by a backslash \, similar to the conventions used in C, C++, and Java.

string-literal:
    "'"⟨characters⟩*"'"

An identifier is any sequence of characters which is not a keyword and matches the regular expression

identifier:
    (*letter*|_)(⟨letter⟩|_|⟨digit⟩)*

For some navigation expressions a literal representing some empty collection is necessary. This literal is called `nil`.

nil-literal:
    `"nil"`

The undefined value is different from `nil`, because it represents a runtime error. In addition to the OCL means of working with the undefined value we introduce a literal for undefined.

undefined-literal:
    `"undefined"`

To summarize, a literal is defined by the following rules:

literal:
    primitive-literal
  | *collection-literal*
  | *tuple-literal*

primitive-literal:
    boolean-literal
  | *integer-literal*
  | *real-literal*
  | *string-literal*
  | *nil-literal*
  | *undefined-literal*

A.2 Common Rules

This section collects the common rules if we parse a file or an expression.

operation-name:
    path-name
  | one of `<` `<=` `>` `>=` `<>` `=` `"and"` `"or"` `"xor"` `"not"`
      `"implies"` `+` `-` `*` `/`

formal-parameter:
    identifier `":"` *type*

type:
    path-name
  | *collection-type*
  | *tuple-type*

collection-type:
    identifier `"("` *type* `")"`

tuple-type:
    identifier `"{"` (*variable-declaration*$_{list}$)$_{opt}$ `"}"`

path-name:
    identifier
  | *path-name* `"::"` *identifier*

variable-declaration:
    variable-declaration-no-init (`=` *expression*)$_{opt}$

variable-declaration-no-init:
    identifier (`":"` *type*)$_{opt}$

A.3 Files

This section defines the concrete syntax of file declarations. These rules are only applicable, if the constraints are read from a seperate file. If the constraints are read from a model, the rules starting with expression have to be used.

file:
    package-declaration file
  | *context-declaration file*
  | ϵ

package-declaration:
    `"package"` *path-name* *constraint-declaration*$_{list}$ `"endpackage"`

constraint-declaration:
    context-declaration *constraint*$_{list}$

context-declaration:
    `"context"` *classifier-context*
  | `"context"` *operation-context*

classifier-context:
    path-name
  | *identifier*`":"` *path-name*

operation-context:
    path-name `"("` (*formal-parameter*$_{list}$)$_{opt}$ `")"`
  | *path-name* `"("` (*formal-parameter*$_{list}$)$_{opt}$ `")"` `":"` *type*

constraint:
    stereo-type *identifier*$_{opt}$ `":"` *expression*$_{opt}$
  | `"def"` *identifier*$_{opt}$ `":"` *let-expression*$_{list}$
  | `"attr"` *variable-declaration*$_{list}$
  | `"oper"` *operation-definition*$_{list}$

stereo-type:
    one of `"inv"` `"post"` `"pre"`
  | `"local"` followed by one of `"inv"` `"post"` `"pre"`
  | `"interface"` followed by one of `"inv"` `"post"` `"pre"`

let-expression:
    `"let"` *identifier* (`"("` (*formal-paramenter*$_{list}$)$_{opt}$ `")"`)$_{opt}$ (`":"` *type*)$_{opt}$
        `"="` *expression*

operation-definition:
    identifier `"("` (*formal-parameter*$_{list}$)$_{opt}$`")"` `":"` *type* (`"="` *expression*)$_{opt}$

A.4 Expressions

This section defines the concrete syntax of OCL and ASO syntax. It is based on OCL 1.4 with minor extensions. We have, for example, added tuples and messages from OCL 2.0. The distinction between local assertions, interface specifications, and general assertions are hard to express in a context free way. We do not express this distinction in the grammar.

expression:
    logical-implies-expression
  | *let-in-expression*

logical-implies-expression:
    logical-xor-expression
  | *logical-implies-expression* `"implies"` *logical-xor-expression*

logical-xor-expression:
    logical-or-expression
  | *logical-xor-expression* `"xor"` *logical-or-expression*

logical-or-expression:
    logical-and-expression
  | *logical-or-expression* `"or"` *logical-and-expression*

logical-and-expression:
    relational-expression
  | *logical-and-expression* `"and"` *relational-expression*

relational-expression:
    add-expression
  | *add-expression relational-operator add-expression*

relational-operator:
    one of $<$ $<=$ $>$ $>=$ $<>$ $=$

add-expression:
    mul-expression
  | *add-expression* + *mul-expression*
  | *add-expression* − *mul-expression*

mul-expression:
    unary-expression
  | *mul-expression* ∗ *unary-expression*
  | *mul-expression* / *unary-expression*

unary-expression:
    primary-expression
  | − *unary-expression*
  | `"not"` *unary-expression*

primary-expression:
    literal
  | `"("` *expression* `")"`

| *simple-property-call*
| *postfix-expression*
| *operation-call*

postfix-expression:
primary-expression `"."` *property-call*
| *primary-expression* `"->"` *property-call*
| *primary-expression* `"^"` *message-call*
| *primary-expression* `"^^"` *message-call*

simple-property-call:
operation-name `"@pre"`$_{opt}$ *qualifiers*$_{opt}$ *poperty-call-params*$_{opt}$

property-call:
operation-name `"@pre"`$_{opt}$ *qualifiers*$_{opt}$ *poperty-call-params*$_{opt}$

property-call-params:
`"("` *declarator*$_{opt}$ (*expression*$_{list}$)$_{opt}$ `")"`

declarator:
(*identifier* (`":"` *type*)$_{opt}$)$_{list}$ (`";"` *identifier* (`":"` *type*)$_{opt}$ `"="` *expression*)$_{opt}$ `"|"`

message-call:
path-name `"("` (*message-call-argument*$_{list}$)$_{opt}$ `")"`

message-call-argument:
`"?"` (`":"` *type*)$_{opt}$
| *expression*

qualifiers:
`"["` *expression*$_{list}$ `"]"`

A.5 Stereotypes

Constraints in UML may be stereotyped. In UML 2.0 the pre-defined stereotypes are “inv”, “post”, and “pre”. We add three other stereotypes to this list:

local If a constraint has the “local” stereotype, then it is required to be a constraint in the local language.

interface If a constraint has the “interface” stereotype, then it is required to be an interface specification.

time If a constraint has the "time" stereotype, then it is required to be a time constraint only. In this case, the constraint may only involve time expression between events. Time constraints are defined in [18].

A.6 Constraint Interchange

We use the standard methods for interchanging constraints on UML models. We maintain two options: Either the constraints are seperate from the model or they are part of the model.

If the constraints are seperated from the model, then they have to be stored in a file. This file has to be a stream of characters in UTF-8 encoding, which can be parsed by a grammar with the start symbol *file*.

If the constraints are part of the model, then they have to be strings in UTF-8 encoding which can be parsed by a grammar with the start symbol *expression*. The strings have to appear as constraints in the XMI representation. The language attribute of the string must be either "OCL" or "ASO". If the language is "OCL", we use OCL 2.0 as the language. If it is "ASO" we use the language defined in this report.

Bibliography

[1] Alcatel, Computer Associates, Enea Business Software, Ericsson, Hewlett-Packard, I-Logix, International Business Machines, IONA, Kabira Technologies, Motorola, Oracle, Rational Software, SOFTEAM, Telelogic, Unisys and WebGain. *Unified Modeling Language: Infrastructure*, September 2002. Version 2.0. Available for download at `http://cgi.omg.org/cgi-bin/doc?ad/02-09-01`.

[2] Alcatel, Computer Associates, Enea Business Software, Ericsson, Hewlett-Packard, I-Logix, International Business Machines, IONA, Kabira Technologies, Motorola, Oracle, Rational Software, SOFTEAM, Telelogic, Unisys and WebGain. *Unified Modeling Language: Superstructure*, September 2002. Version 2 beta R1 (draft). Available for download at `http://cgi.omg.org/cgi-bin/doc?ad/02-09-02`.

[3] Hubert Baumeister, Rolf Hennicker, Alexander Knapp, and Martin Wirsing. OCL component invariants. In Luqi and Manfred Broy, editors, *Proc. Wsh. Montery - Engineering Automation for Software Intensive System Integration*, pages 208–215, Monterey, 2001. U.S. Naval Postgraduate School.

[4] R. Bayer and E. M. McCreight. Organization and maintenance of large ordered indexes. *Acta Informatica*, 1(3):173–189, 1972.

[5] Boldsoft and Rational Software Corporation and IONA and Adaptive Ltd. *Response to the UML 2.0 OCL RfP (ad/2000-09-03)*, June 3 2002. Revised Submission, Version 1.5. Available for download at `http://www.klasse.nl/ocl/ocl-specification-v1-5.zip`.

[6] Achim D. Brucker and Burkhart Wolff. A note on design decisions of a formalisation of the OCL: The view of Freiburg. Technical Report 168, Institut für Informatik, Albert-Ludwigs-Universität Freiburg, Georges-Köhler-Allee 52, 79110 Freiburg, Germany, January 2002.

[7] Stanley N. Burris and H.P. Sankappanavar. *A Course in Universal Algebra*. Graduate Texts in Mathematics. Springer-Verlag, 1981.

[8] María Victoria Cengarle and Andreas Knapp. On the expressive power of pure OCL. Technical Report 0101, Institut für Informatik, Ludwig-Maximilian Universität München, 2001. Available for download at `http://www.pst.informatik.uni-muenchen.de/veroeffentlichungen/TR0101.pdf`.

[9] Tony Clark and Jos Warmer, editors. *Object Modelling with the OCL*, number 2263 in Lecture Notes in Computer Science. Springer-Verlag, 2002.

[10] Steve Cook, Anneke Kleppe, Richard Mitchell, Bernhard Rumpe, Jos Warmer, and Alan Wills. The Amsterdam manifesto on OCL. In Clark and Warmer [9], pages 115–149.

[11] Ole-Johan Dahl. *Verifiable Programming*. Prentice Hall international series in computer science. Prentice Hall, Hertfordshire, 1992.

[12] Werner Damm, Bernhard Josko, Amir Pnueli, Alexandre David, Johann Deneux, and Julien d'Orso. A formal semantics for a UML kernel language. Technical report, OFFIS, 2002. Internal report.

[13] Willem-Paul de Roever, Frank de Boer, Ulrich Hannemann, Jozef Hooman, Yassine Lakhnech, Mannes Poel, and Job Zwiers. *Cocurrency Verification: Introduction to Compositional and Noncompositional Methods*. Number 54 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 2001.

[14] Desmond Francis D'Souza and Alan Cameron Wills. *Objects, Components, and Frameworks with UML: The Catalysis*$^{\text{SM}}$ *Approach*. The Addison Wesley object technology series. Addison Wesley Longman, Inc., 1998.

[15] Dominic Duggan and Adriana Compagnoni. Subtyping for object type constructors. In *Informal proceedings of the Sixth International Workshop on Foundations of Object-Oriented Languages (FOOL)*, January 1999. Available for download at `http://www.cis.upenn.edu/~bcpierce/FOOL/sched6.html` (June 7, 2002).

[16] Joseph Goguen. Order sorted algebra. Technical Report 14, UCLA Computer Science Department, 1978. Semantics and Theory of Computation Series.

[17] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison-Wesley, 1995.

[18] Susanne Graf and Ileana Ober. OMEGA time extensions. Technical report, Verimag, December 2002. Internal report.

[19] Yuri Gurevich. Evolving algebra 1993: Lipari guide. In E. Börger, editor, *Specification and Validation Methods*, pages 9–36. Oxford University Press, 1995.

[20] Jozef Hooman. Extending hoare logic to real-time. *Formal Aspects of Computing*, 6(6A):801–825, 1994.

[21] S.C. Kleene. *Introduction to Metamathematics*. North Holland, 1952.

[22] Donald E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Addison-Wesley, 2 edition, 1998.

[23] Hillel Kugler. Live sequence charts for the uml. Technical report, Weizmann, December 2002. Internal report.

[24] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. JML: A notation for detailed designpreliminary design of JML: A behavioral interface specification language for Java. Technical report, Department of Computer Science, Iowa State University, October 2002. Available for download at `http://www.cs.iastate.edu/~leavens/JML/Documentation/index.html`.

[25] José Meseguer. Membership algebra as a logical framework for equational specification. In Francesco Parisi Presicce, editor, *12th International Workshop on Recent Trends in Algebraic Development Techniques (WADT'97)*, number 1376 in Lecture Notes in Computer Science, pages 18–61, Tarquinia, Italy, June 1997, 1998. Springer-Verlag.

[26] Bertrand Meyer. *Eiffel: The Language*. Prentice Hall, 1992.

[27] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall, 2nd edition, 1997.

[28] Ileana Ober. Definition of the tool exchange format. Technical report, Verimag, 2002. Internal report.

[29] Arnold Oberschelp. *Logik für Philosophen*. Verlag J.B. Metzler, Stuttgart, Weimar, second edition, 1997. In German.

[30] Object Management Group. *OMG Unified Modeling Language Specification*, September 2001. Version 1.4. Available for download at `http://cgi.omg.org/cgi-bin/doc?formal/2001-09-67`.

[31] Object Management Group. *UML™ Profile for Schedulability, Performance, and Time Specification*, March 2002. Available for download at `http://cgi.omg.org/cgi-bin/doc?ptc/02-03-02`.

[32] Omega Consortium. *OMEGA: Correct Developement of Real-Time Embedded Systems in UML Annex 1 (Description of Work)*, October 2001.

[33] Daniel Parnitzke. On formal semantics of object systems with data and object attributes. Technical Report 2001-15, TU Berlin, March 2001.

[34] François Pottier. *Type inference in the presence of subtyping: from theory to practice*. Research report, INRIA, September 1998. Available for download at `ftp://ftp.inria.fr/INRIA/publication/RR/RR-3483.ps.gz` (June 07, 2002).

[35] Mark Richters. *A Precise Approach to Validating UML Models and OCL Constraints*. PhD thesis, Universtät Bremen, 2002. Logos Verlag, Berlin, BISS Monographs, No. 14.

[36] Andy Schürr. A new type checking approach for OCL 2.0? In Clark and Warmer [9], pages 21–40.

[37] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, special edition, 2000.

[38] Mandana Vaziri and Daniel Jackson. *Some Shortcomings of OCL, the Object Constraint Language of UML*, December 1999. Available for download at `http://sdg.lcs.mit.edu/~dnj/pubs/omg.pdf`.

[39] Jos B. Warmer and Anneke G. Kleppe. *The Object Constraint Language: Precise Modeling With UML*. Addison-Wesley, 1998.

Index