Verifying OCL Specifications of UML Models

Tool Support and Compositionality

The Unified Modelling Language (UML) and the Object Constraint Language (OCL) serve as specification languages for embedded and real-time systems used in a safety-critical environment.

In this dissertation class diagrams, object diagrams, and OCL constraints are formalised. The formalisation serves as foundation for a translation of class diagrams, state machines, and constraints into the theorem prover PVS. This enables the formal verification of models defined in a subset of UML using the interactive theorem prover.

The type system of OCL makes writing specifications difficult while the model is still under development. To overcome this difficulty a new type system is proposed, based on intersection types, union types, and bounded operator abstraction.

To reduce the complexity of the model and to increase the structure of the specification, compositional reasoning is used. The introduction of history variables allows compositional specifications. Proof rules support compositional reasoning.

The feasibility of the presented approach is demonstrated by two case-studies. The first one is the "Sieve of Eratosthenes" and the second one is a part of the medium altitude reconnaissance system (MARS) deployed in F-16 fighters of the Royal Dutch Air Force.

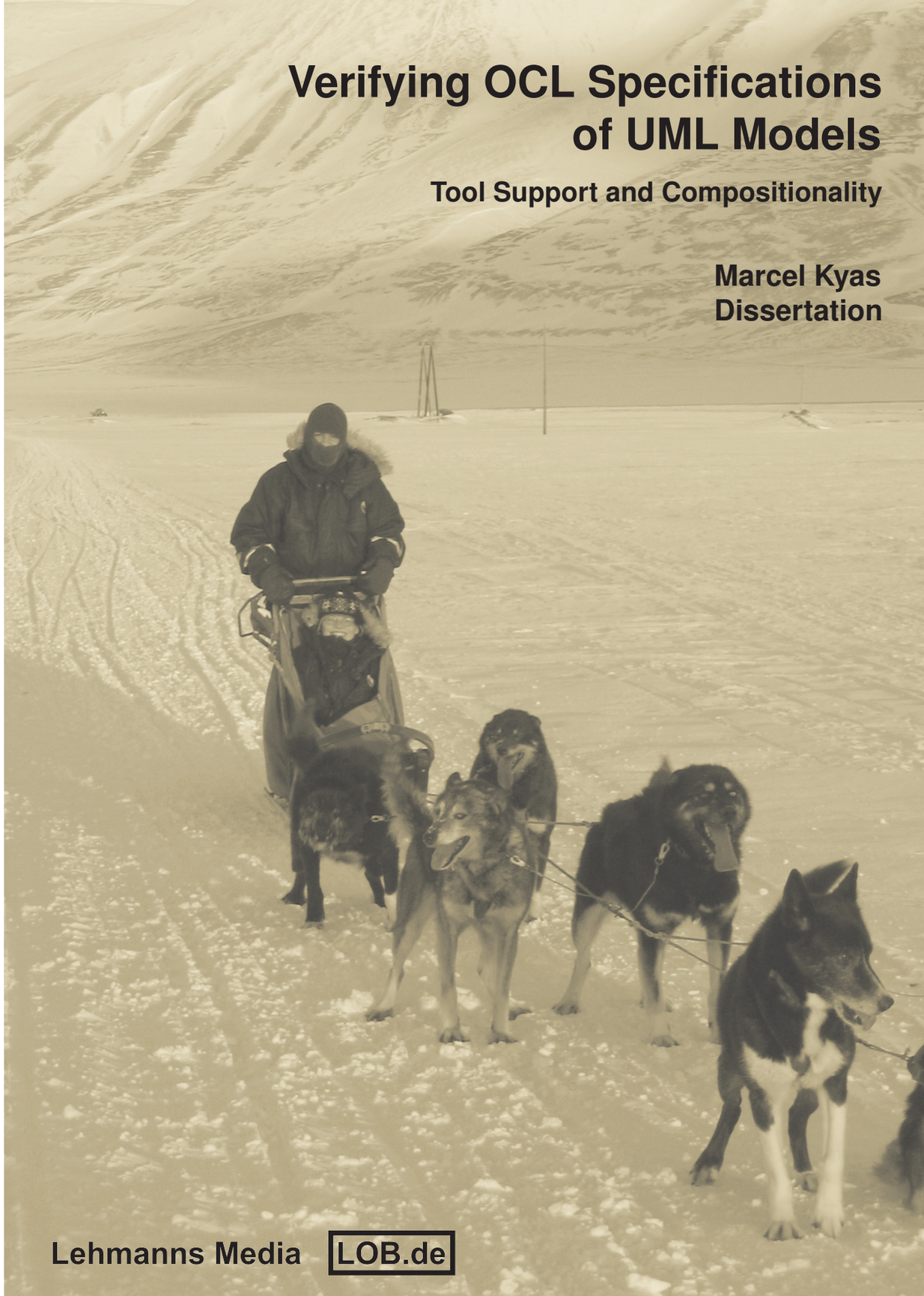
Lehmanns Media LOB.de

ISBN 3-86541-142-8

Marcel Kyas

Verifying OCL Specifications of UML Models

Tool Support and Compositionality



Verifying OCL Specifications of UML Models

Tool Support and Compositionality

Marcel Kyas
Dissertation

Lehmanns Media LOB.de

Verifying OCL Specifications of UML Models: Tool Support and Compositionality

PROEFSCHRIFT

ter verkrijging van
de graad van Doctor aan de Universiteit Leiden,
op gezag van de Rector Magnificus Dr. D. D. Breimer,
hoogleraar in der Faculteit der Wiskunde en
Natuurwetenschappen en die der Geneeskunde,
volgens besluit van het College voor Promoties
te verdedigen op dinsdag 4 april 2006
te klokke 14.15 uur

door

Marcel Kyas
geboren te Pinneberg, Duitsland
in 1975

Promotiecommissie

Promotores:	Prof. dr. J.N. Kok
	Prof. dr. W.-P. de Roever *Christian-Albrechts-Universität zu Kiel*
Copromotor:	Dr. F.S. de Boer
Referent:	Prof. dr. Olaf Owe *Universitetet i Oslo*
Overige leden:	Prof. dr. F. Arbab
	Prof. dr. G. Rozenberg
	Prof. dr. S.M. Verduyn Lunel

Part of this work has been financially supported by IST project OMEGA (IST-33522-2001) and NWO/DFG project Mobi-j (RO 1122/9-1, RO 1122/9-2). The work has been carried out at the Christian-Albrechts-Universität zu Kiel, Germany.

The work in this thesis has been carried out under the auspices of the research school IPA (Institute for Programming research and Algorithmics.

Bibliografische Informationen der Deutschen Bibliothek:

Die Deutsche Bibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über `http://dnb.ddb.de/` abrufbar.

Verifying OCL Specifications of UML Models: Tool Support and Compositionality / Marcel Kyas. – With ref.
Berlin: Lehmanns Media, LOB.de, 2006.
Zugl.: Dissertation, Universiteit Leiden.
Zugl.: IPA Dissertation Series 2006-05.

ISBN: 3-86541-142-8

Printed by docupoint GmbH, Magdeburg, Germany.

Cover photograph "Dog sledging on Svalbard," © 2005 by Martin Steffen. Used with permission.

Preface

Another year is gone;
and I still wear
straw hat and straw sandal.

(Bashō)

This dissertation describes the results of my research at the Chair of Software Technology at the Christian-Albrechts-Universität zu Kiel, which was conducted as part of the Omega project (IST-2001-33522, see also `http://www-omega.imag.fr/`). The aim of Omega was to create a development method in UML (Unified Modelling Language) for embedded and real-time systems built on formal footing. My task was to adapt OCL (Object Constraint Language) to the Omega method. My solution is to extend OCL to a trace-based specification language facilitating compositional reasoning. Especially, I had to solve problems related to object-orientation and object creation.

OCL requires a context in which it is interpreted. This context is provided by other UML diagrams. However, UML is a large language. For this dissertation I have selected class-diagrams and state-machines for providing the context for OCL.

Class diagrams describe the structure of a system. They are well-understood and a very stable part of UML. They were present in the beginning of UML in 1997 and are based on entity-relationship diagrams, introduced in 1976 [24]. Considering this almost thirty year history, class diagrams will probably not change drastically anymore.

State machines provide a notation for describing the behaviour of systems. They evolved from Harel's statecharts [60] and their object-oriented development [61]. According to Hewitt [63], event-driven semantics are the essence of object-oriented computation. Such a semantics is used for state machines and allowed me to avoid all complications of more "modern" object-oriented programming languages.

While I was researching for my dissertation, the UML 2.0 standard has been actively developed. Many promising ideas have been introduced. It was a considerable amount of work to follow these developments and often results became obsolete or invalid. I hope that tracking this moving target contributed to making my results more robust.

Acknowledgements. I thank all people who supported me during the time of researching and writing this thesis.

Bengt Jonsson has been a very insightful reviewer of the Omega project. His opinion

on my deliverables has helped to improve this dissertation. He generously took the time to explain his work and views on trace-based specification and verification of systems.

Willem-Paul de Roever was very attentive of my well-being, sometimes "commanding" me to take a holiday. When I followed his "order" in 2001, I decided to visit Oslo, because at that time my fiancée was studying there. Willem-Paul suggested that I visit Ole-Johan Dahl on this occasion. Instead, I met Olaf Owe who told me that Ole-Johan Dahl was very ill. In spite of me not being invited he took the time to introduce me to his research. I was impressed by his work and suggested that he would present his work in Kiel. Instead he sent Einar Broch Johnsen. This set up the foundation for fruitful collaborations between Oslo, Kiel, and Frank de Boer in Amsterdam. Einar's most memorable contribution was to organise a working meeting on Spitsbergen during February 2005, as displayed on this book's cover.

Frank de Boer proved to be an excellent troubleshooter when we wrote a paper together. He always came up with new ideas whenever a problem arose *and* found the time to discuss them with me.

Martin Steffen's encyclopedic knowledge — not limited to computer science — his attentive observations, and his questions helped me to understand my own ideas better.

I also thank my numerous co-authors: Harald Fecher and Jens Schönborn discussed the subtleties of the semantics of UML; especially Jens worked out the semantics of UML 2.0 state machines, which helped me to understand the version used in the Omega project. Mark van der Zwaag formalised the semantics of Omega state machines in PVS and explained it to me. I based the translation of OCL into PVS on his work. Hillel Kugler and Tamarah Arons were the testers of this translator. Without their feedback, patience, and willingness to work with an evolving tool, many problems with the theory and its implementations would have emerged later, if at all. Jozef Hooman has been a great help in working with PVS and modelling the MARS case study.

Mirco Kuhlmann volunteered to read various drafts of this dissertation. He has pointed out many mistakes and omissions and suggested many improvements.

Our secretaries Sabine Hilge and especially Änne Straßner handled most bureaucratic tasks reliably. Thanks to their effort I was able to focus on research and teaching.

I thank my friends for reminding me of a life with leisure. Especially Jan Engelbach and Ortwin Ebhardt invited me for a chat over lunch or a party whenever the occasion arose.

I thank my parents Heidemarie and Horst for their loving encouragement and support. I shall always remain grateful for their advice.

Finally, my fiancée Ann-Dörte took care of me and most of the daily errants during the final stages in writing this thesis, reminded me to take shorter and longer breaks from work, and was wonderfully supportive and loving. She deserves all the praise for the fact that I remained healthy and sane in this memorable part of my life.

Marcel Kyas
January 26, 2006, Kiel

Contents

List of Tables

List of Figures

Chapter 1

Introduction

This dissertation is concerned with modelling and verification of object-oriented systems, especially object-oriented real-time embedded systems.

Embedded systems usually are computer systems which consist of software and hardware and which are designed to perform accurately defined functions at low cost. Therefore, they are intentionally simplified compared to general-purpose computers. Embedded systems receive their input from sensors and compute a reaction which controls actuators as output. The software deployed in embedded systems is commonly stored in read-only memory or flash memory, which makes it immutable during the lifetime of the system; therefore, it is called *firmware*, because it cannot be easily changed after the system has been deployed. Also, embedded systems are usually deployed outside the reach of people so that the system has to be able to handle various kinds of failures: in case of catastrophic failures, the system usually restarts itself.

Besides being cheap to produce, embedded systems are also expected to run for years without errors. Therefore, firmware is tested more thoroughly than software for personal computers.

Finally, embedded systems interact with the physical world through their input sensors and affect the physical world through their actuators. This interaction of the system is considered part of its functionality. In order to formulate the correct operation and error situations a formal description of the physical world (and the firmware) is required. Of course, the physical world is far to complex to describe, therefore a formal *model* abstracting from irrelevant details has to be developed.

Since the first embedded systems were developed, the complexity of those systems was steadily increasing. The digital autopilot for the Apollo lunar lander required about 2,000 lines of assembly code [27], while it has been estimated that there are about 1.5 million lines of code in the on-board command and control computers on the International Space Station [73].

Object-oriented methods are believed to help improving the structure of large software systems, to increase programmer productivity, and to help in managing software of this size (see any manual on object-oriented design, among others, [136, 71, 11, 98]). *Object-oriented programming* is a paradigm, where a program or software system is composed of a collection of individual units, called *objects*. The objects, which have a

unique identity, act on each other by receiving messages, processing data, and sending messages to other objects. Object-oriented systems are described by *classes*, which are the units of definition of the structure of data and *behaviour* of the system's parts.[1] This behaviour is described in terms of the kind of messages a system's part is accepting and how it processes data. The behaviour is specified in terms of *operations* and is defined in terms of *methods*. At run-time classes are *instantiated* to objects. Objects encapsulate data and expose a public interface through which the data may be queried or manipulated in a safe way.

A class contains a description of the data *(states)* stored in the objects of the class; this data is structured by and accessed through named *attributes*. A class implements its interface by specifying *methods* that describe how the operation specified in its interface are to be performed. Each method specifies only tasks that are related to the stored data.

A class implements one or more *interfaces*. Each interface specifies *operation signatures* of some of the methods of the class. The methods of a class can either be used directly, that is, if the object is known to be an instance of that class, or via one of its interfaces, that is, if the object is only known to implement one of the interfaces.

A class usually describes a set of *invariants* that are preserved by every method in the class. The invariant specifies a constraint on the state of an object that has to be maintained by each of its methods. Additionally, a *precondition* defines constraints which have to be established in order to *invoke* a method. A *postcondition* defines constraints which have to be guaranteed by a method immediately after its execution finishes.

An implementation of a class specifies *constructors* which provide methods that establish the classes invariant during object creation, and fail if the invariant cannot be satisfied.

Example 1.1. Assume a class C, which provides a constructor m, accepting an integer x as parameter. Furthermore, assume that this class has an attribute a and the invariant $a > 0$. Finally, assume that the method implemented for m is $a := x$. Then, whenever the operation m of class C is invoked, a new instance of C is created, provided that the invariant is satisfied at the end of the method body. Consequently, the call $C :: m(1)$ succeeds, because after the completion of the method m the invariant of C is satisfied, and a new object is created. Here, the operator :: means that the operation m of a class C is called statically, that is, not as a property of an object but of the class itself. The call $C :: m(0)$ fails, because the invariant $a > 0$ is violated after the execution of the method; also, no new object has been created. ♦

Additionally, a constructor may be used to acquire the resources needed by an object. Examples are: acquiring memory for parts of the objects, acquiring (or creating) semaphores, acquiring the privilege to use a specific device.

[1] In contrast, *object-based* programming languages define objects but not classes. In such languages objects are not restricted to the class structure, but may change their structure and their interface at run-time.

Additionally, a class may specify at most one *destructor*, whose main purpose is to delete the identity of an object, invalidating any reference to it in the process, and to release all resources used by the instance. In languages supporting garbage collection the destructor cannot be invoked from the program. It will be invoked by the garbage collector if the object becomes unreachable. In this case it is customary to call the destructor *finaliser*. In this thesis we do not consider the explicit deletion of objects but assume the presence of a garbage collector, which frees the resources owned by an object as soon as the last reference to it has disappeared.

1.1 Unified Modelling Language and Object Constraint Language

The Unified Modelling Language (UML) [13, 137] provides a graphical notation for modelling such object-oriented systems. Since its standardisation in 1997 [131] by the Object Management Group (OMG), it has become a commonly accepted notation in industry. Since then the concerns of developers of real-time embedded systems have been taken into account by integrating concepts from the ROOM method [142] and introducing the "UML Profile for Schedulability, Performance, and Time" [107].

UML integrates the concepts of Booch [11], OMT [136], OOSE [71], and Class-Relation by fusing them into a single and widely applicable modelling language for object-oriented systems. UML also aims to be a standard modelling language for concurrent and distributed systems. UML has become an industry standard, created under the auspices of the Object Management Group (OMG).

The Unified Constraint Language (UML) version 2.0 is specified in two documents. The first, entitled "UML 2.0: Infrastructure" [110] specifies the architectural foundations of UML. The second, entitled "UML 2.0: Superstructure" [111] defines the user-level constructs. Both documents define an abstract syntax (which is called *metamodel* in these documents) of UML.

Despite being specified, UML has not yet been developed into a *formal* design language, because lack of formal semantics. A lot of effort has been invested into developing a sound and unambiguous formal semantics for different aspects of UML, for example, in [6, 18, 39, 51, 79, 92, 100, 151], but most of these fail to address how to integrate the formal notations into a software development process (with the dissertation of Alexander Knapp [79] among the exceptions).

UML provides notations for specifying, visualising, constructing, and documenting the artifacts of object-oriented systems. Such a *model* in UML describes the static structure of a software system in terms of *class diagrams*, the behaviour of a software system using *actions*, *state machines*, and *activities*, and its environment using *use cases*. All these notations, however, have not yet been formally defined and there is no agreement on the formal definition of their semantics.

UML 2.0 supports 13 types of diagrams, which can be grouped into two types:

Structural Diagrams Structural diagrams are concerned with the static structure of a system as well as the structure of the system during runtime.

class diagram Central concepts of the static structure of a system are described by classifiers (a *classifier* is anything which may represent a type, for example, classes, interfaces, or data types) and their relationships (like associations and generalisations). Class diagrams focus on time-invariant properties of the system's structure.

object diagram Objects are instances of classes. An object diagram describes a snapshot of the system's state at a particular time. It displays objects, the values of its attributes, and the links connecting objects.

component diagram Component diagrams display the organisation of and the dependencies among a set of components at a high level. Component diagrams address the static implementation view of a system.

composite structure diagrams Composite structure diagrams display the relation between elements working together within a classifier. They are similar to class diagrams and object diagrams, but they display parts and connectors. These parts are not necessarily classes in the model and they need not represent particular instances, but they may represent *rôles* that classifiers play. The composite structure diagram is used to display the runtime architectures of any kind of classifier.

deployment diagram A deployment diagram depicts a static view of the runtime configuration of processing nodes and the components that run on those nodes. In other words, deployment diagrams show the hardware for the system, the software that is installed on that hardware, and the middleware used to connect disparate machines to one another. One creates a deployment diagram for applications that are deployed on several machines. Deployment diagrams can also be created to explore the architecture of embedded systems, showing how the hardware and software components work together. In short, one may want to consider creating a deployment diagram for all but the most trivial of systems.

package diagram Package diagrams are used to display the organisation of packages, that is, named collections of diagrams, and their elements, and provide a visualisation of their corresponding name spaces.

Behavioural Diagrams Behavioural diagrams describe the behaviour of the system or parts of the system.

use case diagram The functionality of a system is described by a set of *use case diagrams* each showing a sequence of interactions between parts of

the system and its environment, represented by *actors*. The focus is on listing actors and the use cases they participate in. Use cases are scenarios described by one sequence of interactions between actors and the system. Separate text is frequently used to describe the use case in detail.

state machine diagram UML state machine diagrams depict the various states that an object may be in and the transitions between those states. States in a state machine are depicted by an enclosed area. In fact, in other modelling languages, it is common for this type of a diagram to be called a state-transition diagram or even simply a state diagram. A state represents a stage in the behaviour pattern of an object, and like UML activity diagrams it is possible to have initial states and final states. An initial state, also called a creation state, is the one that an object is in when it is first created, whereas a final state is one where no transitions exit. A transition is a progression from one state to another and is triggered by an event that is either internal or external to the object.

activity diagram Activity diagrams model the logic captured by a single use case or use scenario, or for modelling detailed logic of a business rule. Although UML activity diagrams could potentially model the internal logic of a complex operation it would be far better to simply rewrite the operation so that it is simple enough that one does not require an activity diagram. In many ways UML activity diagrams are the object-oriented equivalent of flow charts and data flow diagrams from structured software development.

sequence diagram The behaviour of a system can be described by objects exchanging messages. A sequence diagram describes one scenario by the participating objects and the order in which they exchange messages.

interaction overview diagram Interaction overview diagrams display control flow and are variants of UML activity diagrams.

communication diagram Communication diagrams display the flow of messages between objects in an object-oriented application and also imply the basic associations (relationships) between classes.

timing diagram Timing diagrams are used to explore the behaviours of one or more objects throughout a given period of time.

Of all these types of diagrams, only three are used this dissertation. Class diagrams are used to model the static structure of a system and to provide the underlying type information. Object diagrams are used to describe states of systems by providing a snapshot of all objects, their states, and the relation between these objects. Finally, state machines are used for describing the behaviour of systems, that is, how states of systems evolve, respectively the behaviour of the objects which comprise the system.

The UML aims to be an extensible language. It uses *stereotypes* as one of its extension mechanisms. A stereotype is an annotation of an existing model element, which

extends or modifies the meaning of every element which has been *stereotyped* like this. We do not consider the extension mechanism in this thesis, but some predefined stereotypes are used in this thesis. Stereotypes can, at least, be applied to classifiers (which are explained below), operations, signals, and constraints. Examples of stereotypes are «active», which may be applied to classes, «constructor», which may be applied to operations, and «invariant», which may be applied to constraints. The name of a stereotype is written in *guillemets* (« »). In essence, a stereotype is used as a kind of *modifier* of the stereotyped element, which is intended to make the kind, and through this its meaning, more precise.

The Object Constraint Language (OCL) [155] was developed at IBM as a language for business modelling within IBM and is derived from the Syntropy method [33]. It is used in UML both to help formalise the semantics of the language itself and to provide a facility for UML users to express precise constraints on the structure of models. OCL has been available in UML since version 1.1 of the standard. The current version of the OCL standard is 2.0 [113].

OCL is mainly used for expressing properties of a model which cannot be expressed in the diagrammatic notations of UML. It is the standard language in UML allowing the expression of textual and declarative requirements of models in terms of invariants, preconditions, and postconditions. Whether invariants, preconditions, and postconditions have an effect is a *semantic variation point* in the UML standard. One possibility is to ignore these specification. Another one is to abort the program as soon as an invariant, a precondition, or a postcondition is violated. In our examples we chose the operational interpretation.

In order to make UML more accessible for non-experts, the designers of the language decided to prefer natural language over formal notation to explain the concepts of the language. The syntax and the concepts of UML are explained using UML itself, a process called meta-modelling. The *meta-model* of UML is the definition of the abstract syntax of UML using UML's notation. The semantics of these concepts is explained in English, resulting in an informally specified semantics of UML. This may be sufficient for business-modelling in communication in small groups, where an agreement on the meaning of a model is quickly reached between all participants. It is, however, not adequate for communicating between different groups, and for formal verification of models [18].

1.2 Problem Statement

UML is an established modelling language for object-oriented systems. Fuelled by the ROOM-method [142] the notations of UML are also used for real-time embedded systems. A significant number of embedded systems contain safety-critical aspects. There is an increasing awareness in industry of the fact that the application of formal specification languages and their corresponding verification and validation techniques

may significantly reduce the risk of design errors in the development of such systems. If UML is to be applied in the context of real-time or safety-critical systems, which was the goal of the Omega Project,[2] formal semantics of all concepts occurring in a model is mandatory. UML is still lacking such semantics, because it is (intentionally) only informally specified.

Not only is a formal semantics of UML needed, but the modeller has to be enabled to analyse his model and to validate it, preferably using multiple techniques. This requires the development or adaption of analysis, validation, and verification techniques for UML models. If these techniques are to be accepted by modellers in industry, then the developed techniques also require tool support.

In this dissertation formal semantics of a small subset of UML notation is developed, namely class diagrams, object diagrams, and state machines, as well as semantics for OCL. The purpose of this semantics is a translation of models using the formalised notations into the input language of the interactive theorem prover PVS [121]. Then the translated specification may be formally proved correct using PVS.

For proving models with OCL constraints correct, it is necessary to have semantics which makes proving easy by enabling the use of the standard strategies of PVS. It is already hard to learn to use PVS, so a formalisation of OCL in PVS should not require the user to learn additional prover strategies. As a second advantage, the semantics defined here is closer to the mathematical semantics usually taught.

A similar formalisation has already been described by Aredo [6]. The different application domain of Aredo's research lead him to different design decisions. For example, Aredo uses sequence diagrams in his formalism, which we do not use. Instead, we use constraints involving time.

Also Richters formalised the semantics of OCL [133]. His semantics is based on an operational semantics: constraints are evaluated over the simulation or execution of a model. We are more interested in a formal correctness proof. Therefore, semantics had to be developed which enables the validation of constraints in a theorem prover.

Brucker and Wolff propose a semantics of OCL [19] suitable for theorem proving using Isabelle [101]. Their formalisation uses a three-valued semantics, an artifact of the operational interpretation of OCL constraints defined in the standard. While this is desirable if one is interested in proving properties of the *semantics of OCL*, that is, meta-theorems, it quickly becomes a hindrance in proving properties of a concrete *model* and its specification expressed in OCL. In the latter case, one only wants a proof that the constraint is *true*, not whether it evaluates to *false* or *undefined*.

1.2.1 Correctness of Systems

One major concern of all stake holders in software development is that the software should be correct. Today, the complexity of software has exceeded the capabilities of

[2] IST-2001-33522, see `http://www-omega.imag.fr` for more details.

testing alone. Especially concurrent systems are not amenable to testing, because their behaviour is not deterministic, while most testing frameworks assume that a program's behaviour is deterministic. This implies that a test suite which may show an error in a program will not necessarily do so.

Therefore, methods are needed which help in developing correct software. These methods are based on a precise formalism. The main methods applied today are:

Lightweight Verification Lightweight verification refers to the partial verification of program properties. For example, type checking is a light-weight verification method. Also, extended static checking of properties is a form of lightweight verification. Extended static checking refers to the verification of predefined properties of programs without executing it. Also, runtime verification using observers and assertions can be considered to be lightweight verification. Crucial for lightweight verification is that the investment of the user is minimal and that verification is automatic. The disadvantage is that lightweight methods are usually incomplete and sometimes even unsound by reporting false errors.

Model Checking Model-checking provides a more heavy-weight but still automatic verification method by exhaustive state-space exploration. A model-checker expects a finite-state system (the model) and a property expressed in a specification language (usually a temporal logic) and checks whether the model satisfies the property. The appeal of model checking is that if the model does not satisfy the property the model checker will produce a counter example, usually a trace leading to a situation, that is a state for safety properties and a sequence of states describing a cycle for liveness properties, violating the specification. This dissertation is not concerned with model checking.

Program Verification Program verification is the use of formal mathematical techniques for proving that a program satisfies its specification. It is the heaviest technique, because programs are complex mathematical objects, especially if the program is concurrent and object-oriented, as it is the case in the dissertation of Erika Ábrahám [1]. Another important difficulty is formally specifying the system. Proving this specification correct is usually a laborious endeavour when using an interactive theorem prover.

Most people desire that the method is *decidable*, which means that there exists a method, which decides whether $P \models \varphi$ holds. Furthermore, the used method should be sound, that is, whenever the method states that the program has the desired property, then the program indeed has this property. For automatic methods, the user is often willing to accept false negatives, where the tool claims the presence of an error, even though the program is correct (the successful *lint* [74] is a tool with such a behaviour).

Completeness of a proof method means that for any correct property of a system one can also find a proof of its correctness. Most of the time, completeness is only

of theoretical concern. Proving the completeness of a method is useful, if this proof demonstrates a *general method* for proving systems correct. For certification, however, a proof of the desired properties has to be provided, preferably in a format which is checkable by a human. Generally, finding a proof is not trivial, even if we have the guarantee that such a proof exists, which is guaranteed by the completeness of a method. For example, first-order logic has a complete proof theory, but a formal proof cannot be computed automatically for all valid sentences in first-order logic, that is, first-order logic is not decidable. A system specified in first-order logic still has to be verified by *somebody*. Higher-order logic does not even have a complete proof theory, but higher-order logic has been successfully used to specify programs and to prove programs correct.

1.2.2 Compositionality

Compositionality is described by Frege's principle: "The meaning of the whole is a function of the meaning of the parts." In Hoare's proof theory, the parts are a program's syntactic constituents, their statements. Their meaning is characterised by pre- and postconditions. In systems like CSP, the constituents are the *processes*. One advantage of a formalism like CSP is, that a process is also a *syntactic* constituent.

Compositional methods have been successful in verifying concurrent systems. Till now, no truly compositional method has been found for class-based object-oriented systems. In a class-based formalism the only syntactic constituents are *classes*. A class represents a *set* of semantic constituents, namely the *objects*. Therefore, the composition operator, which is syntactically represented in sequential programming languages and in formalisms like CSP, is only a *semantic* operator in class-based object-oriented programming. This implies that we cannot use the syntactic constituents of an object-oriented program directly for compositional reasoning anymore.

While the proof system described by Ábrahám is syntax directed, it is not compositional, because it uses an interference freedom test and a cooperation test [1]. Also, the fundamental composition operator of object-oriented modelling, namely inheritance, has not been covered in her dissertation. On the other hand, it appears to be very hard to find a *compositional* formulation of proof rules for languages with inheritance, because of dynamic binding and open recursion.[3] A Hoare logic for such languages has been described by de Boer and Pierik [127, 128]. Similar to our work they assume closed programs; introducing a new class which inherits from an existing class invalidates all existing proofs. This problem is avoided, if inheritance coincides with behavioural subtyping, as described, among others, by Cusack [37].

The *fragile base class* problem is among the main difficulty in finding compositional proof theories for object-oriented programs. Overriding a method in a subclass may unexpectedly affect the behaviour of methods inherited from a super-class. The fragile

[3] See Section 2.3.5 for a detailed discussion of these concepts.

base class problem has been studied, for example, by Mikhajlov and Sekerinski [99]. Compositional proof methods can be formulated for object-oriented programs, if overriding of methods preserves the behaviour of the original method, that is, implementation inheritance implies behavioural subtyping. This disciplined use entails Liskov's substitution principle [94] and avoids the fragile base class problem. An alternative approach is to separate implementation from interfaces, where subtyping of interfaces implies behavioural subtyping, whereas implementation can be freely inherited, as long as each class implements its interfaces. This approach has been studied by Owe [120] and Johnsen [72].

Similarly, in a concurrent setting, the *inheritance anomaly* gives rise to complications. Inheritance and synchronisation constraints conflict with each other, such that methods have to be redefined in order to maintain a concurrent objects integrity. The term was coined by Matsuoka and Yonezawa [96]. Also our setting is concurrent but we chose state machines for describing the behaviour of objects. Because the meaning of inheritance of state machines is not clear, we decided that state machines are not inherited but behaviour has to be always overridden. Consequently, the inheritance anomaly does not occur in our setting.

1.3 Contribution of this Dissertation

In this dissertation we develop a formal semantics of UML and OCL suitable for proving a system correct using the interactive theorem prover PVS [121]. Instead of investigating a deep embedding in PVS, that is formalising the abstract syntax and its semantics of the used notations in PVS, we preferred a shallow embedding, that is generating correctness conditions in the logic of the theorem prover. While we cannot prove statements *about* the semantics with this approach, our method allows us to focus on proving the correctness of a particular system.

One main concern of the method proposed by us is to allow proving the system correct during early stages of design. This implies that a concrete implementation of the system is not required. One can prove parts of the system correct while no concrete behaviour has been specified by proving specifications correct.

The method proposed here adapts the ideas of mixed formalisms[4] to object-oriented modelling. This is inspired by similar work on untimed systems [114, 115, 158] and related to work on timed systems [65, 139]. The approach combines the results presented in [84], [83], and in [85].

Consistency of the specifications is established by proving the type-correctness of the specification, as we have described in [82]. Establishing type correctness of specifications is, despite of Lamport's opinion [90], a very useful step in verifying the consistency of a specification. If the type system is safe, pointing out type errors in

[4] A mixed formalism is a formalism where specifications and programming constructs can be freely mixed.

the specification also points out an inconsistency of the specification. The type system should, of course, be expressive and flexible enough in that it allows the specifier to focus on the specification and not on the type correctness of his specification.

Finally, in order to prove properties of a model correct efficiently, we had to adapt the semantics of UML and OCL. This is necessary, because the semantics of OCL is specified using an interpreter. Because there exist constraints for which the interpreter need not terminate, an *undefined* value was introduced into the semantics, resulting in a three-valued logic. For the correctness of a system, however, it is irrelevant, whether the interpreter diverges on an OCL specification. Therefore, we propose a *declarative* semantics of OCL for *proving* systems correct with respect to their specification, which is based on the semantics of higher-order logic and which only uses two truth values. We prove that our declarative semantics agrees with the interpreter for all true constraints, except if they involve quantification over infinite collections. A constraint, which involves quantification over infinite collections, may be true even if the interpreter states that it is *undefined*. An abbreviated version of this work has been published in [86].

The definition of UML state machines and its semantics has been derived from the Diploma thesis of Jens Schönborn [140] and the work of Fecher, Kyas, de Boer, de Roever, and Schönborn described in [52] and [53].

1.4 Publication History

The results described in this dissertation have been previously published as:

I. Marcel Kyas and Frank S. de Boer. On message specification in OCL. In Frank S. de Boer and Marcello Bonsangue, editors, *Proceedings of the Workshop on the Compositional Verification of UML Models (CVUML)*, volume 101 of *Electronic Notes in Theoretical Computer Science*, pages 73–93. Elsevier, November 2004.

II. Marcel Kyas, Harald Fecher, Frank S. de Boer, Mark van der Zwaag, Jozef Hooman, Tamarah Arons, and Hillel Kugler. Formalizing UML models and OCL constraints in PVS. In Gerald Lüttgen, Natividad Martínez Madrid, and Michael Mendler, editors, *Proceedings of the Second Workshop on Semantic Foundations of Engineering Design Languages (SFEDL 2004)*, volume 115 of *Electronic Notes in Theoretical Computer Science*, pages 39–47. Elsevier, 2005.

III. Marcel Kyas. An extended type system for OCL supporting templates and transformations. In Martin Steffen and Gianluigi Zavattaro, editors, *Formal Methods for Open Object-Based Distributed Systems: 7th IFIP WG 6.1 International Conference*, volume 3535 of *Lecture Notes in Computer Science*, pages 83–98. Springer-Verlag, 2005.

IV. Marcel Kyas, Frank S. de Boer, and Willem-Paul de Roever. A compositional trace logic for behavioural interface specifications. *Nordic Journal of Computing*, 12(2):116–132, 2005.

V. Marcel Kyas and Jozef Hooman. Compositional verification of timed components using PVS. In Bettina Biel, Matthias Book, and Volker Gruhn, editor, *Software Engineering 2006*, volume P-79 of *Lecture Notes in Informatics*, pages 143–154. Gesellschaft für Informatik e.V., Kollen Verlag, Bonn, 2006.

1.5 Dissertation Outline

This dissertation is organised as follows: Chapter 2 describes the syntax and semantics of OCL 2.0 and a subset of UML 2.0 which is required to explain the semantics of OCL. In order to allow a comparison between our modified semantics of OCL and the semantics defined in the standard, we define the semantics of OCL in terms of an abstract machine which interprets constraints in a pair of states of a model, which is also called *action*. An action is needed to interpret postconditions, which may also refer to the state of the precondition.

Chapter 3 defines an extended type system for OCL which helps overcoming deficiencies of the current type system of OCL. Soundness of this type system and decidability of the type checking problem is proved. This chapter extends [82] with proofs and a careful exposition of the type-checking algorithm.

Chapter 4 describes a translator from the notations introduced in Chapter 2 into the input language of the interactive theorem prover PVS. This chapter is an extended version of [86] and explains the translation of a subset of UML and OCL to PVS.

Chapter 5 describes an extension of the Object Constraint Language allowing trace-based specifications. We demonstrate that our extension allows to express OCL's message expressions, which state that a message has been sent during the execution of a method in its postcondition, but also allows to reason about messages received. This chapter is based on [84].

Chapter 6 describes how to reason compositionally about objects of an UML specification. The idea is based on formalising an object's behaviour in terms of invariants on their communication traces. This chapter is based on [85].

Chapter 7 applies the tools and methods developed in this dissertation to a case study. It contains an extended version of [88].

In Chapter 8 we conclude the dissertation with a short summary and an outline of future work.

Additionally, the symbol □ marks the end of a proof, the symbol ◊ marks the end of a definition, and the symbol ♦ marks the end of an example.

Chapter 2

Introduction to Models of OCL and UML

In this chapter we present the main notations and concepts used in this thesis and introduce the relevant parts of the Unified Modelling Language (UML) [111, 110] and the Object Constraint Language (OCL) [113] used in this thesis. We explain the notions of class diagrams, object diagrams, object constraint language, and define a formal semantics for all these concepts, which are used in later chapters.

In this chapter we assume a non-empty and enumerable infinite set $\mathcal{N}$ of names, which is partitioned into a set $\mathcal{C}$ of class names, $\mathcal{A}$ of attribute names, $\mathcal{M}$ of operation names and $\mathcal{E}$ of association end names. All these notions will be explained in the next section.

2.1 Class Diagrams

The most well-known language of UML is the language of *class diagrams* which describes the structure of a system in terms of classifiers, their interfaces, and the relations between classifiers.

A *classifier* describes a set of instances or values that have features, for example, attributes and operations, in common. *Classes*, *interfaces*, and *data types* are classifiers. A classifier can be generalised[1] by other classifiers.

Example 2.1. The class *OclAny*, described in Section 2.3.4, is a class that generalises every other class in every model. *Integer* and *Real* are data types, where *Real* generalises *Integer*. However, *Integer* and *Real* are not classes, because you cannot create new instances of them. ♦

Specifically, the different classifiers used in this thesis, are:

Class A *class* defines structural features, for example, attributes, and behavioural features, for example, operations. Moreover, classes are required to provide *methods*, that is, implementations, for each operation they define. Classes may be instantiated.

[1] Generalisation is the inverse of inheritance, see Section 2.1.1.

Interface An *interface* defines structural and behavioural features like classes, but methods must not be defined for the operations declared in an interface, but often *constraints* on operations are defined in terms of pre- and postconditions for the operations. Interfaces cannot be instantiated.

Abstract Classes An *abstract classes* is like a class, in that it specifies structural and behavioural features, and it is like an interface, in that it need not provide methods for all behavioural features it specifies. Consequently, abstract classes cannot be instantiated. The crucial difference between an interface and an abstract class is that the abstract class may provide a method, whereas an interface must not.

Data type A *data type* is a type whose instances are only identified by their value. In contrast, instances of a class are identified by their unique name. The most prominent data types in UML are the *primitive data types*, for example, *Boolean*, *Integer*, or *Set*.

By applying the stereotype «active» to a class, the class is declared *active*. Such an *active class* is a class where each instance of this class has its own thread of control. Classes, which are not declared to be active are *passive class*. Passive classes do not have their own thread of control, but their methods are executed by the thread of another active object.

Example 2.2. In the Java Programming Language, each class is passive by default. Objects whose class inherits from the class Thread, are active, after the start method has been called. See [57] for details. ♦

By applying the stereotype «signal» to a classifier, the instances of that classifier may be used as messages, which are sent asynchronously. The message itself is called a *signal*. Only active objects may receive these messages.

A classifier specifies a collection of *features*. Features are classified into *structural features*, which are attributes, and *behavioural features*, which are operations and receptions (a reception declares that an object can receive a signal of a certain type). For the sake of simplicity, we do not allow overloading of operations or generalisation on signals. Overloading of operations can be compiled away by renaming the operation names such that the name of each operation is unique.

In principle UML allows any object direct access to the data stored in an object, where this access is controlled by a *visibility* attribute of the feature. Here, we ignore this attribute. We assume later that all structural features, that is, attributes, are private to the object, that is, every object may access its own attributes only, and behavioural features are public, that is, every object may invoke operations of every other object. However, to constrain the object structure, these attributes and the associations are considered to be public in OCL and, therefore, part of the interface.

Before we formally define interfaces, we introduce the following notation: $A \to B$ designates the type of functions from A to B, whereas $A \rightharpoonup B$ designates the type of

partial functions from A to be B. Observe that any function is also a partial function. We introduce a special predicate $E(f(x))$ which holds, if and only if the partial function f is defined in x. Using this predicate we define for any partial functions f its domain $\text{dom}(f) \stackrel{\text{def}}{=} \{x \mid E(f(x))\}$ and its range $\text{rng}(f) \stackrel{\text{def}}{=} \{f(x) \mid x \in \text{dom}(x)\}$. Each partial function $f : A \rightharpoonup B$ can be extended to a total function $f_\perp : A \to B$, by defining $f_\perp(x) = f(x)$ if $x \in \text{dom}(f)$ and $f(x) = \perp$ otherwise. In the sequel we assume that each partial function f is extended to a total function $f_\perp$ and we write simply f for $f_\perp$. We write A^B for the set of all functions of type $B \to A$.

We describe part of the interface using regular languages and identify words of the language with sequences. The symbol ϵ designates the empty sequence. A set S of letters represents the choice of letters, consequently, we write $\cup$ for the choice between two languages. Concatenation of words and languages is written by juxtaposition, or sometimes using the operator $\cdot$. The Kleene-star is written as $S^\star$.

The interface of a class is specified using *segment descriptors*. A segment descriptor defines the interface defined by a class. The *full descriptor* defines all properties defined by a class, including the properties of all super-classes and super-interfaces (compare to Definition 2.3).

Definition 2.1 (Segment Descriptor)**.** Let $\mathcal{N}$ be a set of names, $\mathcal{C} \subseteq \mathcal{N}$ a set of classifier names, $\mathcal{S} \subset \mathcal{C}$ a set of signal names, $\mathcal{A} \subseteq \mathcal{N}$ a set of names which we call attribute names, $\mathcal{M} \subseteq \mathcal{N}$ a set of names which we call operation names, such that $\mathcal{C}$, $\mathcal{A}$, and $\mathcal{M}$ are pairwise disjoint.

An *operation signature* s is an element of the language characterised by the regular language $\mathcal{C}^\star \cdot (\mathcal{C} \cup \{\text{void}\})$. The last classifier in s describes the return type of the operation, where void states that the operation does not return a value.

The *segment descriptor* of a classifier c is the union of:

- A function $c_\mathcal{A} : \mathcal{A} \rightharpoonup \mathcal{C}$ which is a partial function defining the attributes defined in the class and the type of the attributes.[2]
- A function $c_\mathcal{M} : \mathcal{M} \rightharpoonup (\mathcal{C}^\star \cdot (\mathcal{C} \cup \{\text{void}\}))$ describing the operations and the signature of the operation the class provides.
- If c is active, then the set $c_\mathcal{S} \subseteq \mathcal{S}$ assigns the class a set of signals which it accepts.

A segment descriptor can be interpreted as a partial function $c : \mathcal{N} \rightharpoonup (\mathcal{C} \cup (\mathcal{C}^\star \cdot (\mathcal{C} \cup \{\text{void}\})) \cup \{\text{true}\})$ assigning attribute names to their types, operation names to their signatures, and signal names to true, which indicates that instances of the classes which are defined by this segment descriptor accept this signal. ◊

Two types (or classifiers) are equal in UML, if they have the same name. This means that typing in UML is by names. Therefore, two objects are of a different type, if the

[2] A Function is also used as a set of pairs which describes its graph.

class they have been instantiated from have different names, even if all properties of the objects are identical. One exception to this rule is the typing of *tuples* in OCL, which are anonymous classes without operations. Two tuple-types are equal if they have the same members of the same type. This means, that typing of tuple types is by structure.

The relations between classes are *generalisation*, and *association*.

2.1.1 Generalisation

Generalisation captures the hierarchical relationship between classes. For example, "mammal" generalises "dog", "horse", "cat" and many others. Because a "dog" *is a* "mammal" it *inherits* all features of a mammal. Consequently, the *generalisation* relation is the inverse of the inheritance relation commonly defined in object-oriented programming languages. Formally, generalisation is a *partial order* $\preceq$ on classes C.[3] If a class $c \in C$, which we call parent, *generalises* another class c', which we then call child, written $c' \preceq c$, then all features defined by c are also features defined by c'. This means that the parent of a class has actually less features than its child. In Chapter 3 we describe how c' may override a feature of c in a type-safe way.

A class diagram describes the signature of classes in a distributed manner using *segment descriptors*, which define the attributes and operations particular to the class being described. The *full descriptor* contains the segment descriptor of the current class and, recursively, the full descriptor of all its parent classes. This way inheritance of features is realised.

The UML standard defines the following constraints for a class hierarchy.

1. For a classifier, no attribute, operation, or signal with the same signature may be declared in more than one of the segments (in other words, they may not be redefined).

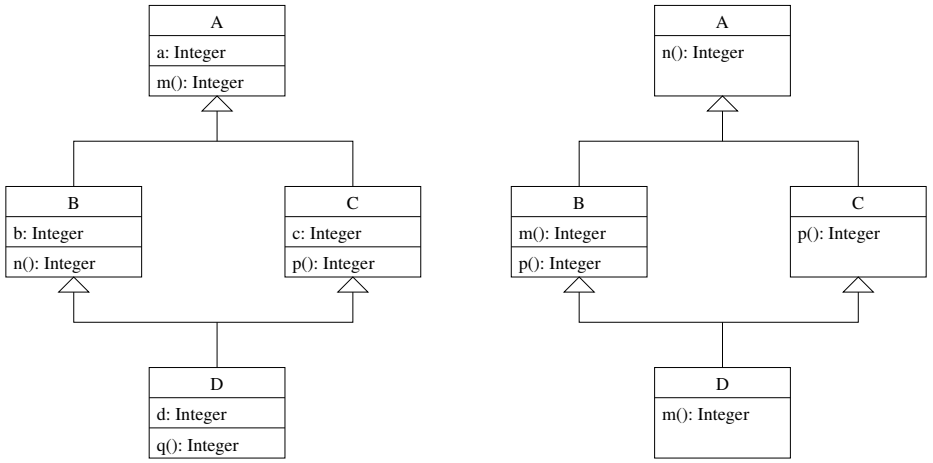
2. Methods may be declared in more than one segment. Use the *youngest*. If there are two methods of the same age, the diagram is ill-formed.

3. The constraints on the full descriptor are the union on the segment current class and all of its ancestors. If any of them are inconsistent, then the model is ill-formed.

For any partial order $\leq$ define $\uparrow x \stackrel{\text{def}}{=} \{y \mid x \leq y\}$ and $\downarrow x \stackrel{\text{def}}{=} \{y \mid y \leq x\}$. Using these notations we define generalisations, taking the three properties above into account:

Definition 2.2 (Generalisation)**.** Let $\preceq$ be a partial order on classifier names C. The relation $\preceq$ is called a *generalisation relation*, if:

[3] A *partial order* R is a relation that is *reflexive*, that is, $\forall x : xRx$, *transitive*, that is, $\forall x, y, z : xRy \wedge yRz \implies xRz$, and anti-symmetric, that is, $\forall x, y : xRy \wedge yRx \implies x = y$.



(a) Valid Generalisation (b) Invalid Generalisation

Figure 2.1: Valid and invalid generalisations

1. $\forall c, c' : c' \preceq c \wedge c \neq c' \implies c \cap c' = \emptyset$, that is, c generalises c' when $c = c'$ or the segment descriptors of the class being generalised are disjoint from the segment descriptor of the generalising class.

2. $\forall c, c', c'' : c \preceq c' \wedge c \preceq c'' \wedge c \neq c' \wedge c \neq c'' \wedge c' \neq c'' \implies c' \cap c'' = \emptyset$, that is, if c is generalised by two different classifiers c' and c'', then the segment descriptors of c' and c'' are disjoint. ◊

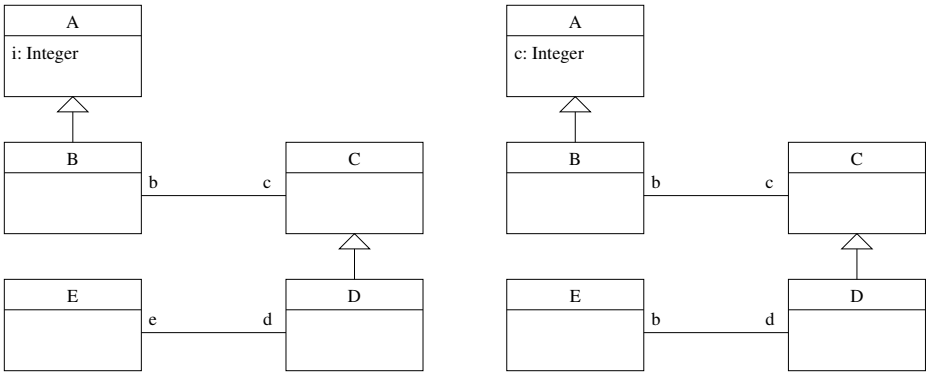
Figure 2.1 displays two class diagrams with generalisations. A generalisation is displayed by an arrow between two classes, where the generalised class is at the end without any triangle, and the generalising class is at the end with the triangle. Consequently, both class diagrams claim that $B \preceq A$, $C \preceq A$, $D \preceq B$, and $D \preceq C$.

The class diagram (a) on the left hand side of Figure 2.1 displays generalisations, which all conform to Definition 2.2. All features defined in the segment descriptors are unique to their segment descriptor.

The class diagram (b) on the right hand side of Figure 2.1 displays generalisations which violate Definition 2.2. There, the class B does not generalise class D, because both define the method m in their segment descriptor. This contradicts Condition 1 of Definition 2.2. Moreover, D cannot be generalised by B and C, because both B and C define an operation p. This contradicts Condition 2 of Definition 2.2

Definition 2.3 (Full Descriptor)**.** Let $\preceq$ be a partial order on classes and c a classifier. Then the *full descriptor* of c is defined as: $(\!|c|\!) \stackrel{\text{def}}{=} \bigcup_{c' \in \uparrow c} c'$. ◊

In the sequel we omit the parenthesis $(\!|c|\!)$ and simply write c for the full descriptor of a class, if this causes no ambiguity.



(a) Valid Associations (b) Invalid Associations

Figure 2.2: Valid and invalid associations

2.1.2 Association

Associations define relations on instances of classifiers. Associations are inspired by relations from relational databases. Associations consist of named ends referring to classifiers. An association with n ends is a n-ary relation on objects of the classifiers involved in the association. The names of association ends are treated like attributes of the objects participating in an association. Moreover, associations are *navigable*, that is, they provide names under which an object o "knows" another object o'.

Definition 2.4 (Association)**.** Let C be a set of classifiers. An *association end* is a pair $\langle e, c \rangle$, where $e \in \mathcal{E}$ is a name and $c \in C$ is a class. An association is a set r of association ends satisfying:[4] For any association $r = \{\langle e_1, c_1 \rangle, \langle e_2, c_2 \rangle, \ldots, \langle e_n, c_n \rangle\}$ with $n \geq 2$, whose graph describes a function, we have: $\forall i : \forall c : 1 \leq i \wedge i \leq n \wedge c \in \bigcup_{1 \leq j \leq n} \downarrow c_j \implies (\!|c|\!)(e_i) = \bot$. ◊

By this definition the association end names occurring in an association are not allowed to be part of the full descriptor of the participating classes and their subclasses. Furthermore, in the definition of a class diagram we ensure that all associations do not introduce any conflicts.

Figure 2.2 displays two class diagrams with generalisations and associations. An association is displayed by a line between two classifiers.

The class diagram (a) on the left hand side of Figure 2.2 displays two associations, which all conform to Definition 2.4. Association $F = \{\langle b, B \rangle, \langle c, C \rangle\}$ does not conflict with any property defined in the classes A, B, C and D. It also does not conflict with the association $G = \{\langle e, E \rangle, \langle d, D \rangle\}$, because even though D "inherits" an association from C, the association end names do not conflict.

[4] Association names are typically called r from set $\mathcal{R}$ to stress that they are relations between objects and to avoid ambiguity with attributes, typically called a from set $\mathcal{A}$.

The class diagram (b) on the right hand side of Figure 2.2 displays associations which violate Definition 2.4. There, the class B inherits an attribute c from class A, but the association $F = \{\langle b, B\rangle, \langle c, C\rangle\}$ uses the name c for one of its ends, which contradicts the definition.

Finally, we define a class diagram.

Definition 2.5 (Class Diagram)**.** A class diagram D is a triple $(\mathcal{C}, \mathcal{R}, \preceq)$, where $\mathcal{C}$ is a set of classes, $\mathcal{R}$ is a set of association relations over $\mathcal{C}$ and $\preceq$ is a generalisation relation on $\mathcal{C}$. Furthermore, we require the following well-formedness condition on associations in class diagrams:

For all $r, r' \in \mathcal{R}$ with $r \neq r'$ and all $c, c' \in \mathcal{C}$ with $c' \preceq c$ and $e, e' \in \mathcal{E}$ such that $\langle e, c\rangle \in r$ and $\langle e', c'\rangle \in r'$ we require that there exists not $e'' \in \mathcal{E}$ and no c'', c''' such that $\langle e'', c''\rangle \in r$ and $\langle e'', c'''\rangle \in r'$. This means that the association end names of different associations which share related classifiers are different. ◊

The class diagram (b) on the right hand side of Figure 2.2 displays associations violating Definition 2.5: associations $F = \{\langle b, B\rangle, \langle c, C\rangle\}$ and $G = \{\langle b, E\rangle, \langle d, D\rangle\}$ conflict, because both associations have an end called b and an end of type C by way of generalisation, which contradicts the definition. Note that in the latter case, it is ambiguous whether the instances associated under the name b to instances of D are instances of B or E.

2.1.3 Parameterisation

Also occurring in class-diagrams are *parameterised classes*. Parameterised classes are called *templates* in C++ [146] and *generics* in Java [57].

A parameterised class is not a classifier, but it is an *operator* mapping types and values to new types. The standard parameterised class occurring in almost all programming language is the array type. The type ARRAY in itself is not a type. Most of the time, a new array is declared as, for instance, a : ARRAY 10 OF *Integer*, which indicates that ARRAY is an operator taking two arguments: The first argument is the *size* of the array, here 10, and the second argument is the type of the elements stored in the array, here *Integer*.

2.2 Object Diagrams

As described in the previous section, a class diagram specifies the static structure of a model. Object diagrams describe a state or snap-shot of a system during runtime. An object diagram is a semantic model of a class diagram. A pair of object diagrams, which describe an action, is a semantic model for an OCL constraint.

Intuitively, an object diagram describes a state (or part of a state) of a system. It displays objects, the state of each object, and the *links* between objects.

Definition 2.6 (Object State)**.** The *state of an object* is defined by a partial function $\omega : \mathcal{N} \rightharpoonup \mathbb{V}$, where $\mathcal{N}$ is the set of attribute names and $\mathbb{V}$ is a set of values. The set of all object states is called Ω. ◊

Example 2.3. Let $\mathcal{N} = \{a, b, c\}$ be a set of names, and let $\mathbb{V} = \mathbb{B} \cup \mathbb{R}$ be the set of values, which consists of the disjoint union of booleans and reals. Then $\omega = \{a \mapsto \text{true}, c \mapsto 5.5\}$ is an object state, where the attribute a has the value $\omega(a) = \text{true}$ and the attribute c has the value $\omega(c) = 5.5$, whereas no value for the attribute b is defined. ♦

Definition 2.7 (Object Type)**.** Let $\mathbb{O} \subseteq \mathbb{V}$ be an enumerable set of *object identities* with the special object identity $null \in \mathbb{O}$ representing the *absent object*. Let D be a class diagram. The function $\tau : \mathbb{O} \rightharpoonup C$ assigns to object identities a class. Let $o \in \mathbb{O}$ be an object identifier. If $\tau(o)$ is defined, we say that o exists.

In the preceding definition we do not require a type for each object identity. Those object identities, which do not have a type are the identities of those objects, which have not been created yet.

2.2.1 Associations and Navigation Expressions

Next, we define the meaning of associations with the definition of the latter given by Definition 2.4. Recall that an association r maps association end names to classes.

Definition 2.8 (Association Valuation)**.** Let r be an association. Then the semantics of r, called its *valuation*, written $[\![r]\!]$ is a multi-set of partial functions $\text{dom}(r) \rightharpoonup \mathbb{O} \setminus \{null\}$ such that for all $f \in [\![r]\!]$ and for all $n \in \text{dom}(r) \cap \text{dom}(f)$ we have $\tau(f(n)) \preceq r(n)$. ◊

We write $\wr \cdot \int$ to construct multi-sets (or bags).

Observe that an association is not like an attribute; it is more like a table in relational databases, as developed by Codd [29].

Example 2.4. Recall the class diagram depicted in Figure 2.2 (a). Let $\mathbb{O} = \{1, 2, 3, 4, 5\}$. Define $\tau = \{1 \mapsto A, 2 \mapsto B, 3 \mapsto C, 4 \mapsto D, 5 \mapsto E\}$. One association valuation for the association $F = \{\langle b, B\rangle, \langle c, C\rangle\}$ is given by $[\![F]\!] = \wr\{b \mapsto 2, c \mapsto 3\}\int$, associating the object identified by 2 to the object identified by 3. Another association valuation is $[\![F]\!] = \wr\{b \mapsto 2, c \mapsto 3\}, \{b \mapsto 2, c \mapsto 4\}\int$, which associates the object identified by 2 to the object identified by 3 and the object identified by 4. Observe, that this is a correct association state, because $D \preceq C$ in the underlying class diagram. Finally, to provide a counter example, consider the interpretation $[\![F]\!] = \wr f \int$, where $f = \{b \mapsto 1, c \mapsto 3\}$. This interpretation is not a valid association valuation, because $\tau(f(b)) = A$, $F(b) = B$, and $\neg\tau(f(b)) \preceq F(b)$. ♦

Next, we define the interpretation of navigation expressions, for which we already use the syntax of OCL.

Definition 2.9 (Navigation)**.** Let $\mathcal{N}$ be a set of association end names with typical element n. A navigation expression is inductively defined by the grammar $e ::= self \mid e.n$.

The meaning of a navigation expression e in a class diagram $D = \langle C, \mathcal{R}, \preceq \rangle$ is defined as a function $[\![\cdot]\!]$, mapping each navigation expression to a function $\mathbb{O} \to \mathbb{N}^{\mathbb{O}}$, which maps each object identity, used to evaluate the navigation expression *self*, to a bag of objects:[5]

$$[\![e]\!](s) \stackrel{\text{def}}{=} \begin{cases} \wr s \int & \text{, if } e = self \\ \wr o \mid \exists r, f, n' : r \in \mathcal{R} \wedge n \neq n' \wedge f \in [\![r]\!] \\ \qquad\qquad \wedge\, o = f(n) \wedge f(n') \in [\![e']\!](s) \int & \text{, if } e = e'.n. \end{cases}$$

◊

Assuming that all association end names occur in the navigation expression, which type-checking can establish, as described in Chapter 3, we observe that the result of a navigation expression is always a multi-set of objects. In contrast to the usual programming languages the absent object, here denoted by *null*, does not occur in the semantic value of a navigation expression; instead, *null* is represented by the empty bag. Also, it is possible to navigate the empty bag. Navigating the empty bag results in the empty bag.

In contrast to standard OCL, our definition of the meaning of a navigation expression is flattening, that is, navigating the result of a navigation expression results in a bag of objects, and not in a bag of bags. Thereby we retain the semantics of OCL prior to version 2.0.

Example 2.5. Consider the class diagram of Figure 2.3 with classes $C = \{A, B, C, D, E\}$ and associations $X = \{\langle a, A\rangle, \langle b, B\rangle, \langle c, C\rangle\}$, $Y = \{\langle c, C\rangle, \langle e, E\rangle\}$, and $Z = \{\langle b, B\rangle, \langle d, D\rangle\}$.

Let the set of objects be $\mathbb{O} = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$ with $\tau = \{1 \mapsto A, 2 \mapsto A, 3 \mapsto B, 4 \mapsto C, 5 \mapsto C, 6 \mapsto C, 7 \mapsto D, 8 \mapsto D, 9 \mapsto E, 10 \mapsto E\}$. Let the valuations of X, Y, and Z be as displayed in Table 2.1. Then:

$$[\![self.c.e]\!](1) = \wr o \mid \exists y, f, n' : e \neq n' \wedge f \in [\![y]\!] \wedge o = f(e) \wedge f(n') \in [\![self.c]\!](1) \int \quad (2.1)$$

To evaluate this, we have to evaluate:

$$[\![self.c]\!](1) = \wr o \mid \exists x, f', n'' : c \neq n'' \wedge f' \in [\![x]\!] \wedge o = f'(c) \wedge f'(n'') \in [\![self]\!](1) \int \quad (2.2)$$

Since the only association end which is mapped to 1 in Table 2.1 is a, belonging to the association X, apparently $n'' = a$ and $x = X$. So choose $n'' = a$ and $x = X$. Returning to (2.2) we now have:

$$[\![self.c]\!](1) = \wr o \mid \exists f' : f' \in [\![X]\!] \wedge o = f'(c) \wedge f'(a) \in [\![self]\!](1) \int$$

[5] The set of all bags can be represented by the function from its elements into the natural numbers, therefore we write $\mathbb{N}^{\mathbb{O}}$ for the bag of all objects.

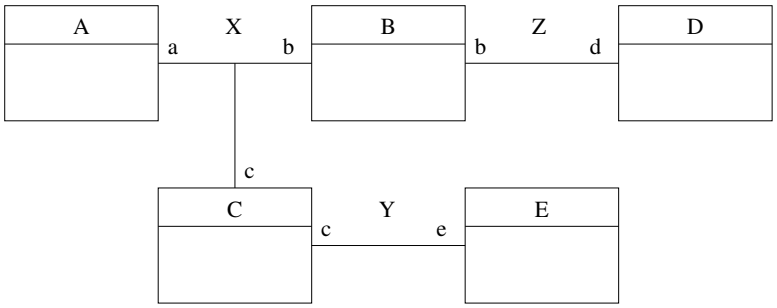


Figure 2.3: Navigation example

$[\![X]\!]$		
a	b	c
1	3	5
1		6
2	3	6

$[\![Y]\!]$	
c	e
4	9
5	9
5	10
6	10

$[\![Z]\!]$	
b	d
4	7
4	7

Table 2.1: Example valuations of associations

In Table 2.1 we find two possibilities for f' with $f'(a) = 1$. For these $f'(c) = 5$ and $f'(c) = 6$ holds. So

$$[\![self.c]\!](1) = \wr 5, 6 \int \tag{2.3}$$

Substituting (2.3) in (2.1) results in:

$$[\![self.c.e]\!](1) = \wr o \mid \exists y, f, n' : e \neq n' \wedge f \in [\![y]\!] \wedge o = f(e) \wedge f(n') \in \wr 5, 6 \int \int$$

From Table 2.1 one finds that $y = Y$ and $o = 9$ or $o = 10$ for $f(c) = 5$ and $o = 10$ for $f(c) = 6$. So, $n' = c$ and one concludes that

$$[\![self.c.e]\!](1) = \wr 9, 10, 10 \int$$

♦

Example 2.6. Reconsider Example 2.5. We now evaluate $[\![self.b.c]\!](7)$.

$$[\![self.b.c]\!](7) = \wr o \mid \exists x, f, n' : c \neq n' \wedge f \in [\![x]\!] \wedge o = f(c) \wedge f(n') \in [\![self.b]\!](7) \int \tag{2.4}$$

To evaluate this, we have to evaluate:

$$[\![self.b]\!](7) = \wr o \mid \exists z, f', n'' : b \neq n'' \wedge f \in [\![z]\!] \wedge o = f'(b) \wedge f'(n'') \in [\![self]\!](7) \int \tag{2.5}$$

This association end is mapped to 7 in Table 2.1 is d in association Z, therefore $n'' = d$ and $z = Z$. Returning to (2.5) we now have:

$$[\![self.b]\!](7) = \wr o \mid \exists f' : f \in [\![Z]\!] \wedge o = f'(b) \wedge f'(d) \in [\![self]\!](7) \int \tag{2.6}$$

We find two possibilities in the association valuation of Z with $f'(d) = 7$, which both have $f'(b) = 4$. Therefore, Equation (2.5) simplifies to:

$$[\![self.b]\!](7) = \wr 4, 4 \int \tag{2.7}$$

Substituting (2.7) in (2.4) results in:

$$[\![self.b.c]\!](7) = \wr o \mid \exists x, f, n' : c \neq n' \wedge f \in [\![x]\!] \wedge o = f(c) \wedge f(n') \in \wr 4, 4 \int \int \tag{2.8}$$

Now we have to find an association where one end is 4 and the other end has the name c. Unfortunately, no association in Table 2.1 satisfies this. Note that association Y is not eligible, because 4 is in column c, but this is the association end we want to navigate. Therefore, we simplify (2.8) to

$$[\![self.b.c]\!](7) = \wr o \mid \text{false} \int$$

which is the empty bag. ♦

This last example demonstrates that any navigation expression is always defined.

We now define the state of the system, which is called *object diagram* in UML, as the state of all its objects, and an association valuation.

Definition 2.10 (Object Diagrams, Links)**.** An *object diagram* $\sigma = \langle \tau, \xi, [\![\cdot]\!] \rangle$ is a triple that consists of a partial function $\tau : \mathbb{O} \rightharpoonup C$ assigning to each object identity $o \in \mathbb{O}$ its class, a partial function $\xi : \mathbb{O} \rightharpoonup \Omega$, where the set of all these functions is called Ξ, mapping object identities to object states, which satisfies for all $o \in \mathbb{O}$ that $\tau(o)$ is defined if and only if $\xi(o)$ is defined, and a function $[\![\cdot]\!]$ assigning to each association an association valuation. The set of Object Diagrams is denoted by Σ.

A *link* named n exists between two objects in the object diagram, written $o \overset{n}{\twoheadrightarrow} o'$, if $o' \in [\![self.n]\!](o)$. If there exists a n such that $o \overset{n}{\twoheadrightarrow} o'$, we write $o \twoheadrightarrow o'$. ◊

Observe that $\twoheadrightarrow$ is symmetric.

Proposition 2.11. *Let $\langle \tau, \xi, [\![\cdot]\!] \rangle$ be an object diagram. Then $\twoheadrightarrow$ is symmetric.*

Proof. Let o, o' be a pair of objects such that $o \twoheadrightarrow o'$. By Definition 2.10 we then find an association r and an association end name n such that $o' \in [\![self.n]\!](o)$. By Definition 2.9 we find a name n' and an association state f such that $f \in [\![r]\!]$ and $f(n) = o'$ and $f(n') = o$. Consequently, we have $o \in [\![self.n']\!](o')$, and therefore $o \twoheadrightarrow o'$. □

It is possible to encode associations as attributes of the type of the associated classifier if each object identifier, except for *null*, occurs at most once in each column. We prove this in the next lemma.

Lemma 2.12. *Let $\sigma = \langle \tau, \xi, [\![\cdot]\!] \rangle$ be an object diagram, where in any valuation of each association an object is, for each association end name, associated to at most one other object. Then the association can be represented by a set of attributes, with the respectively associated classifier as the attribute's type, named by the association ends of the object.*

Proof. Let $\sigma = \langle \tau, \xi, [\![\cdot]\!] \rangle$ be as described in the assumption and D the underlying class diagram. First, we prove that under this assumption $\twoheadrightarrow$ can be interpreted as a function with the signature $f : \mathbb{O} \to \mathcal{E} \to \mathbb{O}$, where this function results in *null* if there exists no object o' such that $o \overset{e}{\twoheadrightarrow} o'$.

Let o be an object in σ and e an association end name. Then, by assumption, there exists at most one object o' with $o \overset{e}{\twoheadrightarrow} o'$. If such an object does not exist, we assign $f(o, e) = \mathit{null}$. Assume that such an o' exists. If there are two distinct objects o' and o'' such that there are e and e' with $o \overset{e}{\twoheadrightarrow} o'$ and $o \overset{e'}{\twoheadrightarrow} o''$, then Definition 2.5 and the assumption imply $e \neq e'$. Therefore, $\twoheadrightarrow$ is functional.

To prove that the association can be represented by a set of attributes, with the respectively associated classifier as the attribute's type, named by the association ends of the object, let D' be the class diagram obtained from D by removing all associations but adding the association end names as attributes of the target's classifiers type to the associations' source classifiers. Observe, that by Definition 2.5 D is a well-formed class diagram. Similarly, from σ we obtain an object diagram σ' on D', where each association end e, which now is an attribute, of o is interpreted by $\xi'(o)(e) = \iota x : o \overset{e}{\twoheadrightarrow} x$. □

We use Russell's ι operator for *definite description*. The expression $\iota x : P(x)$ reads as "The uniquely determined x such that $P(x)$ holds. If no such x exists ore it is not unique, the meaning of the expression $\iota x : P(x)$ is not defined. However, in the context of $\twoheadrightarrow$ we assume that if no object o' exists, such that $o \overset{e}{\twoheadrightarrow} o'$ holds, then $\iota x : o \overset{e}{\twoheadrightarrow} x$ shall represent *null*.

2.2.2 Relating Object Diagrams to Class Diagrams

In order to relate an object diagram to a class diagram we introduce two relations between class diagrams and object diagrams. The first relation formalises that an object diagram conforms to a class diagram, when the values assigned to the objects in the object diagram comply with those in the class diagram. This notion of conformance is very useful for object diagrams which display a view on the state of a system, that is, which abstract from certain details.

Definition 2.13. An object diagram $\sigma = \langle \tau, \xi, [\![\cdot]\!] \rangle$ conforms to a class diagram $D = \langle C, \mathcal{R}, \preceq \rangle$, written $\sigma \triangleright D$, if and only if:

- For all objects $o \in \mathbb{O}$ we have $\tau(o) \in C$, that is, all objects in an object diagram have a type defined in the class diagram.

- For all objects $o \in \mathbb{O}$ and all attribute names $a \in \mathcal{A}$ where $\xi(o)(a)$ is not undefined it holds that $\tau(\xi(o)(a)) \preceq \tau(o)(a)$, that is, whenever an object o defines a value for a name a, then the classifier of this object defines an attribute for this name with a type that conforms to the type of the value of a in o.

- For all objects $o \in \mathbb{O}$ and $o' \in \mathbb{O}$ with $o \rightarrowtriangle o'$ we find an association $r \in \mathcal{R}$ satisfying:

  - There exists $e \in \mathcal{E}$ with $\xi(o)(e) = o'$ and there exists a classifier $c' \in C$ with $\tau(o') \preceq c'$ such that $\langle e, c' \rangle \in r$.
  - If there exists a $e' \in \mathcal{E}$ and a classifier $c \in C$ with $\tau(o) \preceq c$ and $\langle e', c \rangle \in r$, then $o' \rightarrowtriangle o$. ◊

The empty object diagram, that is, $\sigma = \langle \emptyset, \emptyset, \emptyset \rangle$ conforms to all class diagrams.

Example 2.7. Consider the class diagram in Figure 2.1 (a). Assume there is only one object with identity 1 with $\tau(1) = B$. For $\xi = \{1 \mapsto \{a \mapsto 1\}\}$ (because instances of B inherit the attributes of all their super-classes) and an empty association valuation, we obtain a valid object diagram. However, for $\xi = \{1 \mapsto \{a \mapsto \text{true}\}\}$ the object diagram does not conform to the class diagram in Figure 2.1. ♦

The second relation formalises whether the object diagram displays all information defined in a class diagram. Such object diagrams completely describe a state conforming to a class diagram.

Definition 2.14. An object diagram $\sigma = \langle \tau, \xi, [\![\cdot]\!] \rangle$ is a state of a class diagram $D = \langle C, \mathcal{R}, \preceq \rangle$, written $\sigma \models D$, if and only if:

- $\sigma \triangleright D$.

- For all $c \in C$ and $o \in \mathbb{O}$ with $\tau(o) = c$ we have for all $n \in \mathcal{N}$: $\tau(\xi(o)(n)) \preceq c(n)$, that is, for every object o of class c it holds that the type of the value $\xi(o)(n)$ of its attribute n is a subtype of the type of the attribute declared in the segment descriptor of the class. ◊

Example 2.8. Consider again the class diagram in Figure 2.1 (a). Assume there is only one object with identity 1 with $\tau(1) = B$. For $\xi = \{1 \mapsto \{a \mapsto 1\}\}$ and an empty association valuation this does not define a state for the class diagram. For $\xi = \{1 \mapsto \{a \mapsto 1, b \mapsto 0\}\}$ and an empty association valuation this defines a state for the class diagram. ♦

These are all definitions needed for working with object diagrams.

2.3 Object Constraint Language

In this section we present the syntax of OCL, describe the syntactic categories, and develop the minimal syntactic categories of OCL. Well-typedness of OCL terms is described in Chapter 3.

The important notions developed in this section define when a constraint φ holds within a particular object diagram σ which conforms to its class diagram D. This is written as: $D, \sigma \models \varphi$.

Some constraints are behavioural constraints, that is, a pair of a precondition φ and a postcondition ψ. For such specifications we need to evaluate the constraints in a pair σ, σ' of object diagrams, also called an *action* This is written as: $D, \sigma, \sigma' \models \langle \varphi, \psi \rangle$.

The question whether a constraint φ *conforms* to a class diagram D, written as $\varphi \triangleright D$, is a type-checking problem and is the topic of Chapter 3, that is, describing when φ is well-typed in D.

2.3.1 Context of Constraints

All constraints in OCL are associated with a *context*. The context of a constraint defines the entities for which the constraint has to hold. It also defines the way it is interpreted. Context declarations link constraints to class diagrams. OCL 2.0 allows classifiers, attributes, and operations as contexts for constraints. In this section we define the abstract syntax of the *context declaration*, whereas we define the abstract syntax of the constraints in the following Section 2.3.2. Syntactically, we assign a list of constraints φ to its context c using the form:

$$\textbf{context}\ c\ (s : \varphi)^\star \quad .$$

Each constraint φ is tagged with s, which stands for one of **inv**, **pre**, **post**, **init**, **deriv**, and **def**. These tags are called stereotypes in UML. The intended meaning of these stereotypes is:

- Constraints of stereotype **inv** are *invariants*.
- Constraints of stereotype **pre** are *preconditions*.
- Constraints of stereotype **post** are *postconditions*.
- Constraints of stereotype **init** are *initial value specifications*, usually attached to attributes.
- Constraints of stereotype **deriv** are *derived value specifications*, that is, they specify a rule defining how the value of an association or an object's attribute is to be computed.

- Constraints of stereotype **def** are indeed no constraints but definitions of derived attributes and side-effect-free operations which are only used in OCL constraints. These definitions supersede the let-construct from OCL 1.4. The type of the term in a **def** context has to conform to the return type of the operation or attribute being defined.

We only consider constraints with stereotypes **inv**, **pre**, and **post**. Expressions of stereotypes **init** are used to define the initial value of an attribute, whereas expressions of stereotype **deriv** define how the value of a derived attribute or association is computed. These stereotypes do not define when a system is correct but are part of the specification of the system. Therefore, we do not consider these expressions here.

Classifier

One context of an OCL constraint is the *classifier context*. Only invariants are allowed as constraints on classifiers. A constraint in the context of a classifier c is written as

$$\textbf{context}\ c\ \textbf{inv}:\ t \quad ,$$

where c is the name of the context and t is an OCL expression.

Operation

Another context of an OCL constraint is the *operation context*. In this context only pre- and postconditions are allowed, which are used to specify the behaviour of the constrained operation. A constraint in the context of an operation m defined in c is written as

$$\begin{array}{l}\textbf{context}\ c :: m(v_1 : T_1, \ldots, v_n : T_n) : T \\ \textbf{pre} : t \\ \textbf{post} : t'\end{array} \quad ,$$

where c is the class which defines the operation m with formal parameters $v_1, \ldots, v_n$ of types $T_1, \ldots, T_n$, respectively, and where the operation returns a value of type T. The constraint t describes the precondition, whereas t' describes the postcondition of m. If either of them is omitted, then we define it to true.

2.3.2 Abstract Syntax

We start with a description of abstract OCL, a simple core language, into which almost all OCL expressions can be translated. The grammar of OCL is defined next. First, we define the set of all OCL types, and then we define the set of all OCL expressions.

Definition 2.15 (Grammar of OCL Types)**.** Let $D = \langle C, \mathcal{R}, \leq \rangle$ be a class diagram.

1. Every $c \in C$ is an OCL Type.

2. Every data type defined in the OCL standard library (see Section 2.3.4), that is, *OclAny*, *OclVoid*, *OclInvalid*, *Boolean*, *Integer*, *Real* is an OCL type.

3. If τ is an OCL type, then $Collection(\tau)$, $Set(\tau)$, $Bag(\tau)$, $Sequence(\tau)$ are OCL types.

4. If $v_0, \ldots, v_n$ are variable names and $\tau_0, \ldots, \tau_n$ are types, then $Tuple\{v_0 : \tau_0, \ldots, v_n : \tau_n\}$ is an OCL type. ◊

Definition 2.16 (Grammar of OCL)**.** Let $\mathcal{V}$ denote an enumerable set of names we call *variables*, which contains *self* and *result*.

1. true and false are OCL expressions. null is an OCL expression.

2. $\ldots, -1, 0, 1, \ldots$, that is, each integer number is an OCL expression.

3. If $e_1, e_2, \ldots, e_n$ is a list of OCL expressions, which may be empty (in which case $n = 0$), then $Bag\{e_1, e_2, \ldots, e_n\}$, $Sequence\{e_1, e_2, \ldots, e_n\}$, and $Set\{e_1, e_2, \ldots, e_n\}$ are OCL expressions. Expressions of this kind are called *collection literal expressions*.

   If e, e' are expressions, then $e..e'$ is a *range expression*. Range expressions may only occur in collection literal expressions.

4. Any variable name $v \in \mathcal{V}$ is an OCL expression.

5. If t, t_1, t_2 are OCL expressions, so is **if** t **then** t_1 **else** t_2 **endif**. We call expressions of this kind *conditional expressions* (of course, t is intended to be of type *Boolean*, but in this definition syntax is separated from type-checking).

6. Let $n \in \mathbb{N}$. If t, t_i are OCL expressions, v_i variable names, and T_i are OCL types, for all $i \leq n$, then **let** $v_0 : T_0 = t_0, \ldots, v_n : T_n = t_n$ **in** t is an OCL expression. We call expressions of this type *let expressions*.

7. If t is an OCL expression and v is an attribute name, then $t.v$ is an OCL expression. We call expressions of this kind *attribute call expressions*.

8. If t is an OCL expression and v is an attribute name, then $t.v$@**pre** is an OCL expression. We call expressions of this kind *previous attribute call expressions*. This syntax is only allowed in postconditions and is intended to refer to previous values defined by the precondition in a pre-post specification.

9. If $t_0, t_1, \ldots, t_n$ are OCL expressions and m is an operation name, then an *operation call expression* is written as $t_0.m(t_1, t_2, \ldots, t_n)$.

10. If $t_0, t_1, \ldots, t_n$ are OCL expressions and n is an operation name, then an *previous operation call expressions* is written $t_0.v$@**pre**$(t_1, t_2, \ldots, t_n)$. This syntax is only allowed in postconditions.

11. For OCL expressions $t_0, t_1, \ldots, t_n$ and for m an operation name $t_0 \rightarrow m(t_1, t_2, \ldots, t_n)$ is an OCL expression. We call expressions of this kind *collection property call expressions*.

12. An *iterate expressions* is written $t_0 \rightarrow \mathit{iterate}(v_0 : T_0, v_1 : T_1, \ldots, v_n : T_n; a : T = t \mid t_n)$, where t, t_0, t_n are OCL expressions, $a, v_0, v_1, \ldots, v_n$ are variable names, and $T, T_0, T_1, \ldots, T_n$ are types.

    The intention of an iterate expression is to apply the *body* t_n, in which the *accumulator* a and the *iterator variables* $v_0, v_1, \ldots, v_n$ occur freely, to each element of the collection t_0, where each iterator variable visits each member of the collection itself. The result of the iterate expression is accumulated in a, whose value is initialized with the value of t.

13. If t is an OCL expression, so is $t \rightarrow \mathit{flatten}()$. We call expressions of this kind *flatten expressions*. In principle, flatten expressions are a kind of collection call expressions, but we introduce them as atomic expressions, because they are treated specially in the type system. See Section 3.3.5 for details.

14. $T.\mathit{allInstances}()$ and $T.\mathit{allInstances}@\mathbf{pre}()$ are OCL expressions for all type expressions T. These expressions are called *all instances of* T, respectively, *all previous instances of* T expressions. Because of the special semantics of this kind of expression and because it is applied to a type name, we introduce this expression here.

We call the set of expressions generated by this grammar $\mathcal{L}_{\mathrm{OCL}}$. ◊

This inductive definition of the abstract syntax of OCL can be summarised by the following grammar given in BNF (see Appendix A for an explanation of our notation).

$$
\begin{aligned}
t ::= {} & \mathrm{true} \mid \mathrm{false} \mid \cdots \mid -1 \mid 0 \mid 1 \mid \cdots \mid \mathit{self} \mid v \mid t.a \mid t.m(t_1, \ldots, t_n) \\
& \mid t \rightarrow m(t_1, \ldots, t_n) \mid t \rightarrow \mathit{iterate}(v_0 : \tau_0, \ldots, v_n : \tau_n; a = t \mid t_0) \\
& \mid t \rightarrow \mathit{flatten}() \mid \mathbf{if}\ t\ \mathbf{then}\ t'\ \mathbf{else}\ t''\ \mathbf{endif} \\
& \mid \mathbf{let}\ v_0(v_{0,0}, \ldots, v_{0,m_0}) : \tau = t_0, \ldots, v_n(v_{n,0}, \ldots, v_{n,m_n}) : \tau = t_n\ \mathbf{in}\ t
\end{aligned}
$$

The preceding definition only characterises a subset of all OCL expressions. A grammar of the concrete syntax of OCL is described in Appendix A. Other expressions, for example, *forAll*, have been omitted, because they can be translated into the core language defined above. From now on we only assume well-typed expressions, where well-typedness of OCL expressions is defined in Chapter 3.

An OCL expression is called an *OCL formula* or *OCL constraint* if the type of the constraint in the current contextual class-diagram is *Boolean*. In the next section we define a formal semantics for OCL constraints based on the OCL standard [113] and the thesis of Richters [133], and discuss the problems of this semantics.

2.3.3 Semantics

We develop a semantics of OCL as specified in Section 2.3.2 by specifying an *abstract machine with oracles* that interprets OCL constraints in a class diagram and two object diagrams representing an action. This abstract machine is a formalisation of the evaluator described in the OCL 2.0 standard [113, Chapter 10]. In the standard the semantics is defined using UML diagrams and OCL constraints, whereas we define a function which interpreting constraints. A similar interpreter has been proposed by Mark Richters in [133] for OCL 1.3.

The purpose of this interpreter is to define a reference semantics of OCL which we can use to translate OCL constraints into PVS and to identify classes of constraints which cannot be represented in PVS. The translation of OCL to PVS and its soundness proof is described in Chapter 4.The second purpose is to establish a foundation on which the merits and flaws of OCL can be discussed.

Before we formally define the abstract machine, we have to point out that this machine is not computationally effective, because OCL defines a function *oclIsInvalid* which is supposed to result in true on diverging expressions and which therefore solves the halting-problem. Therefore, we first define an abstract machine for the fragment of OCL *without oclIsInvalid*. The semantics full OCL is then defined by introducing an *oracle* which decides whether *oclIsUndefined* returns true or false.

Another crucial observation is the way recursive specifications may be written. The variable *result* refers to the result of an operation call in the postcondition of the operation. Its value can be specified recursively, as described in [113, p. 31]:

> 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.

One of the changes from OCL 2.0 to its preceding versions is forbidding recursive let-definitions [113, p. 104]. Instead, recursive expressions in OCL are defined by defining new side-effect-free auxiliary operations in the class diagram:

> Note that let-definitions that take arguments are no longer allowed in OCL 2.0. This feature is redundant. Instead, a modeller can define an additional operation in the UML Classifier, potentially with a special stereotype to denote that this operation is only meant to be used as a helper operation in OCL expressions. The postcondition of such an additional operation can then define its result value.

The OCL 2.0 standard fails to define the meaning of such a recursive expression. One way to specify the meaning of such an expression is by constraining *result* in the operations post-condition, as described above. However, it is not defined what the meaning of such an expression is if the pre-condition does not hold.

The second solution would be to define a method for the operation, which is recursive. The method need not be implemented in OCL. It can be specified in any other programming language, one of the behavioural notations of UML, or even using natural language. It is not our intention to formalise these different ways to specify a method body.

Furthermore, the operation, which occurs in an OCL constraint can be specified by its post-condition *and* a method. Then it is not specified whether the meaning described by the post-condition is to be used or whether the method is to be used for interpreting the constraints.

Recall that the definition provided in this section is intended to provide the basis on which we can prove the soundness of our translation and that the abstract machine defined in this section is intended as a formalisation of the interpreter defined in the OCL 2.0 standard using UML and OCL. Later, we are only concerned with the translation of OCL constraints in PVS. These translated constraints have to be type consistent in PVS, which implies that termination of recursive functions has to be proved and that each function is only applied to values where it is defined.

In order to simplify the presentation of the translation from OCL to PVS, our definitions adapt the definition of the type universe used to define the formal semantics of the logic used in the theorem prover PVS [123] to our purposes.

In the following definition of the type universe we use for each carrier D, which interprets a type of OCL, the special symbol $\mathcal{J}_D$, which denotes the undefined value for the carrier D. The value $\mathcal{J}$ is a value like any other, and subject to the same rules of reasoning. For example, given two terms t and t', which both represent $\mathcal{J}$, we may conclude $t = t'$ is true. The values $\mathcal{J}$ represent run-time errors like division by zero, the failure of the abstract machine to proceed with its evaluation, or that the interpretation of an expression by the abstract machine may not terminate. However, OCL treats the values $\mathcal{J}$ as proper values in its interpreter (even though it does not have a corresponding literal in the concrete syntax), because, for example, one may test whether an expression evaluates to $\mathcal{J}$ using *oclIsUndefined*.[6] Keep in mind that type consistency in PVS implies that all expressions are defined, that is, their value is not $\mathcal{J}$ (see Chapter 4, where we show that definedness of the translated constraint implies the definedness of the original constraint.).

Definition 2.17 (Type Universe)**.** The *type universe* U_ω of OCL is inductively defined by:

- $\mathbb{B}_\mathcal{J} \stackrel{\text{def}}{=} \{\text{true}, \text{false}, \mathcal{J}_\mathbb{B}\}$ as interpretation of the boolean literals, where $\mathcal{J}_\mathbb{B}$ repre-

[6] We specifically use $\mathcal{J}$ instead of $\perp$ (read "bottom") to avoid confusion with $\perp$ as it appears in domain theory. The main reason for this are that the semantics of OCL is given as an operational semantics and not as a denotational semantics. Also, in denotational semantics, $\perp$ is supposed to mean non-termination or "no value at all", whereas $\mathcal{J}$ *is* a value. Also, the definition of *oclIsUndefined* in OCL defines a non-monotonic and non-computable function. Therefore, classical domain theory is not applicable for the semantics.

sents the undefined truth value.

- $\mathbb{R}_{\mathcal{J}} \stackrel{\text{def}}{=} \mathbb{R} \cup \{\mathcal{J}_{\mathbb{R}}\}$ as interpretation of numeric expression, where $\mathcal{J}_{\mathbb{R}}$ represents the undefined real number.
- $\mathbb{O}_{\mathcal{J}} \stackrel{\text{def}}{=} \mathbb{O} \cup \{\mathcal{J}_{\mathbb{O}}\}$ as interpretation of objects identities, where $\mathcal{J}_{\mathbb{O}}$ represents the undefined object. Observe that $\mathcal{J}_{\mathbb{O}}$ is distinct from $null \in \mathbb{O}$, where *null* will be the interpretation of null.
- $U_0 = \{\mathbb{B}_{\mathcal{J}}, \mathbb{R}_{\mathcal{J}}, \mathbb{O}_{\mathcal{J}}\}$ is the type universe of the individual.
- For each $i \in \mathbb{N}$ define $U_{i+1} = U_i \cup \{(X \times Y) \cup \{\mathcal{J}_{X \times Y}\} \mid X, Y \in U_i\} \cup (\bigcup_{X \in U_i}(2^X \cup \{\mathcal{J}_{2^X}\})) \cup \{X^Y \cup \{\mathcal{J}_{X^Y}\} \mid X, Y \in U_i\}$.
- Finally define $U_\omega \stackrel{\text{def}}{=} \{\mathcal{J}\} \cup \bigcup_{i \in \mathbb{N}} U_i$. ◊

This means that for any level $i \in \mathbb{N}$ the set U_{i+1} contains all values defined on the level i together with the power set of all elements of this level, the set of all functions between elements of level i, and the set of all pairs of elements of level i. We assign to each type occurring in the OCL specification an element of U_ω.

Definition 2.18 (Type Interpretation)**.** We define a function $\mathfrak{D}$ from OCL types to elements of U_ω as follows:

- $\mathfrak{D}(Boolean) \stackrel{\text{def}}{=} \mathbb{B}_{\mathcal{J}}$.
- $\mathfrak{D}(Integer) \stackrel{\text{def}}{=} \mathbb{R}_{\mathcal{J}}$, or to be precise, $\mathfrak{D}(Integer) \stackrel{\text{def}}{=} \mathbb{Z}_{\mathcal{J}}$, where $\mathbb{Z}_{\mathcal{J}} \stackrel{\text{def}}{=} \mathbb{Z} \cup \{\mathcal{J}_{\mathbb{R}}\}$. Note that $\mathbb{Z}_{\mathcal{J}} \subset \mathbb{R}_{\mathcal{J}}$ holds.
- $\mathfrak{D}(Real) \stackrel{\text{def}}{=} \mathbb{R}_{\mathcal{J}}$.
- $\mathfrak{D}(Bag(\tau)) \stackrel{\text{def}}{=} \mathbb{N}^{\mathfrak{D}(\tau)} \cup \{\mathcal{J}_{\mathbb{N}^{\mathfrak{D}(\tau)}}\}$.
- $\mathfrak{D}(Set(\tau)) \stackrel{\text{def}}{=} 2^{\mathfrak{D}(\tau)} \cup \{\mathcal{J}_{2^{\mathfrak{D}(\tau)}}\}$.
- $\mathfrak{D}(Sequence(\tau)) \stackrel{\text{def}}{=} \{\mathcal{J}_{\mathfrak{D}(\tau)^{\mathbb{N}}}\} \cup \bigcup_{n \in \mathbb{N}}(\mathfrak{D}(\tau)^{\{x \in \mathbb{N} \mid x < n\}}$, that is, $\mathfrak{D}(Sequence(\tau))$ is the set of functions from the set of the first n natural numbers into $\mathfrak{D}(\tau)$.
- Methods are treated as ordinary functions, which are curried for sake of simplicity. Given a function of $\tau \to \zeta$, define $\mathfrak{D}(\tau \to \zeta) \stackrel{\text{def}}{=} \mathfrak{D}(\zeta)^{\mathfrak{D}(\tau)} \cup \{\mathcal{J}_{\mathfrak{D}(\zeta)^{\mathfrak{D}(\tau)}}\}$. ◊

The type interpretation assigns to each type defined in OCL an element of U_ω in order to interpret that type. For example, we assign to the type $OclAny \to Boolean$, which represents the type of methods defined in class *OclAny* that do not accept any further arguments and return a boolean value, the set ${\mathbb{B}_{\mathcal{J}}}^{\mathbb{O}_{\mathcal{J}}}$.

For any function $f : A \to B$ we define its *variant* $f\{y \mapsto z\}$, where $y \in A$ and $z \in B$, to be:

$$f\{y \mapsto z\}(x) \stackrel{\text{def}}{=} \begin{cases} f(x) & \text{, if } x \neq y \\ z & \text{, if } x = y \end{cases}$$

and write $f\{y_1 \mapsto z_1, y_2 \mapsto z_2\}$ for $f\{y_1 \mapsto z_1\}\{y_2 \mapsto z_2\}$, provided $y_1 \neq y_2$. The empty function, that is, the function which is not defined for any argument, is written as ϵ.

For defining the semantics of an OCL expression we further assume an interpretation function $\mathcal{I} : U_\omega \to \Sigma \to \mathcal{N} \to U_\omega \to U_\omega$[7] which provides the meaning of the primitive functions defined in the standard library and the methods defined in the class diagram.[8] The intuitive meaning of $\mathcal{I}$ is:

1. For some $o \in \mathbb{O}$ (recall that $\mathbb{O} \subseteq U_\omega$, here, we restrict to objects) the expression $\mathcal{I}(o)$ represents a function which assigns to each operation name $\mathcal{N}$ defined in the class of o a signature. In classic implementation of object-oriented programming languages this represents the so-called *virtual table*, which is used to bind operations to its method.

2. Because the state of an object is defined as part of the global state of the system, the expression $\mathcal{I}(o)(\sigma)$ provides the state in which the method is to be evaluated.

3. The expression $\mathcal{I}(o)(\sigma)(n)$ represents a function, that is, an element of U_ω, taking the arguments provided to the operation call and returning some value. These arguments do *not* include the *self* parameter, whose value is o and can be considered to be already bound because it was originally used to obtain the virtual table. Furthermore, the function is required to be type-consistent.

4. The function $\mathcal{I}(o)(\sigma)(n)$ is assumed to result $\mathcal{J}_T$, where T is the declared result type of the operation, if this function is either undefined for its argument, or if it is diverging.

5. If n is defined in OCL using the **def** stereotype, then $\mathcal{I}(o)(\sigma)(n)(v)$ is required to be the evaluation of the OCL expression associated to n, where the formal parameters are interpreted with the values of the actual arguments given to the call.

This interpretation function is not only used to encode late-binding of the self parameter. Another purpose of this function is to provide an *oracle* which evaluates all calls and functions which are not defined in OCL. Here, we do not define an operational

[7] The signature of $\mathcal{I}$ is actually $\mathcal{I} : U_\omega \to \Sigma \to \mathcal{N} \to U_\omega$. However, we want to emphasise that the result of $\mathcal{I}(o)(\sigma)(m)$ is supposed to be a method whose interpretation is $\mathcal{I}(o)(\sigma)(m) : U_k \to U_\ell$ for some $k, \ell \in \mathbb{N}$. Then there also exists an i such that $i > k$ and $i > \ell$ with $\mathcal{I}(o)(\sigma)(m) : U_i$. In order to simplify the notation, we write $U_\omega \to U_\omega$.

[8] The interpretation of the meaning of primitive operations provided in the OCL standard library is defined in Section 2.3.4.

semantics the possible languages in which methods and operations are defined or all usable behavioural notations of UML. All these notations may be used to define the meaning of operations which can be invoked from OCL expressions.[9] Because these calls may not terminate, the interpretation function is also an oracle which can decide termination. As a consequence, the abstract machine for OCL constraints in the following definition cannot be implemented.

In the next definition, we need the following auxiliary function, which is used to interpret iterate expressions, where $X \in U_\omega$, $x \in 2^X$, $f : X \to Y \to Y$ (f is used to compute the next accumulator value from the currently chosen element and the current accumulator value) for some $Y \in U_\omega$ and $a \in Y$:

$$\text{iterate}(x, a, f) \stackrel{\text{def}}{=} \begin{cases} \mathcal{J}_{2^X} & \text{, if } x \text{ is not finite} \\ a & \text{, if } x \text{ is empty} \\ \text{iterate}(x \setminus \{\varepsilon y : y \in x\}, & \\ \qquad f(\varepsilon y : y \in x, a), f) & \text{, otherwise} \end{cases} , \tag{2.9}$$

where the term $\varepsilon x : P(x)$ uses to Hilbert's *ε-function*, which selects a x satisfying $P(x)$, or is not defined if no such x exists (see, for example, Bourbaki [15]). Similar definitions can be given for multi-sets and sequences.

Now we define the abstract machine used to interpret an OCL constraint. This abstract machine is recursively defined by a function

$$\text{eval} : \mathcal{D} \times \Sigma \times \Sigma \times (\mathcal{V} \to \mathbb{V}) \to \mathcal{L}_{\text{OCL}} \to \mathbb{V} \quad .$$

In any invocation of $\text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(\varphi)$ the parameter D defines the class diagram over which the constraint is to be interpreted, σ and $\overleftarrow{\sigma}$ are two object diagrams over D forming an action, η is a valuation of the local variables, and φ is the constraint to be interpreted. Furthermore, we assume that all types of OCL and its operations are defined in D, and that the constraint is well-typed in D, as described in Chapter 3.

Definition 2.19 (Semantics of Expressions)**.** Let D be a class diagram and let $\overleftarrow{\sigma} = \langle \overleftarrow{\tau}, \overleftarrow{\xi}, \overleftarrow{[\![\cdot]\!]} \rangle$ and $\sigma = \langle \tau, \xi, [\![\cdot]\!] \rangle$ be two object diagrams forming an action, such that $\overleftarrow{\sigma} \models D$ and $\sigma \models D$, and $\mathcal{I} : U_\omega \to \Sigma \to \mathcal{N} \to U_\omega \to U_\omega$ be a function interpreting all methods defined in the class diagram (which contains the OCL standard library). Then the semantics of OCL expressions is defined by:

1. $\text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(\text{true}) \stackrel{\text{def}}{=} \text{true}$ and $\text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(\text{false}) \stackrel{\text{def}}{=} \text{false}$.
   $\text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(\text{null}) \stackrel{\text{def}}{=} \mathit{null}$
2. For any numerical literal ℓ define $\text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(\ell) \stackrel{\text{def}}{=} \ell$.

[9] The only constraint imposed is that evaluating the call is free from side-effects, that is, it does not change the state in which it is evaluated.

3. For collection literal expressions define:

   a) For *Bag*, define

$$\begin{aligned}\text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(Bag\{e_1, e_2, \ldots, e_n\}) \stackrel{\text{def}}{=} \\ \biguplus \{\!| \text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(e_i) \mid 1 \leq i \leq n \text{ and} \\ e_i \text{ is not a range expression} |\!\} \cup \\ \bigcup \{\!| \text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(e_i) \mid 1 \leq i \leq n \text{ and} \\ e_i \text{ is a range expression} |\!\} \quad ,\end{aligned}$$

   where

$$\begin{aligned}\text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(e..e') \stackrel{\text{def}}{=} \\ \biguplus \{\!| i \mid \text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(e) \leq i \leq \text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(e') |\!\} \quad .\end{aligned}$$

   b) For *Set*, the definition is analogous.

   c) For *Sequence*, define

$$\begin{aligned}\text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(Sequence\{\}) &\stackrel{\text{def}}{=} \langle\rangle \\ \text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(Sequence\{e_0, e_1, \ldots, e_n\}) &\stackrel{\text{def}}{=} \langle \text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(e_0) \rangle \cdot \\ &\quad \text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(Sequence\{e_1, \ldots, e_n\}) \\ \text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(e..e') &\stackrel{\text{def}}{=} \text{range}(\text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(e), \text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(e'))\end{aligned}$$

   where,

$$\text{range}(x, y) \stackrel{\text{def}}{=} \begin{cases} \langle\rangle & \text{, if } x > y \\ \langle x \rangle \cdot \text{range}(x + 1, y) & \text{, otherwise.} \end{cases} \quad .$$

4. For any $v \in \mathcal{V}$ define $\text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(v) \stackrel{\text{def}}{=} \eta(v)$. Observe that type checking guarantees that this value is always defined, as demonstrated in Chapter 3.

5. For if-expressions:

$$\begin{aligned}&\text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(\textbf{if } t \textbf{ then } t_1 \textbf{ else } t_2 \textbf{ endif}) \stackrel{\text{def}}{=} \\ &\qquad \begin{cases} \text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(t_1) & \text{, if eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(t) = \text{true} \\ \text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(t_2) & \text{, if eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(t) = \text{false} \\ \mathcal{J} & \text{, otherwise.} \end{cases}\end{aligned}$$

6. $\text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(\textbf{let } v = t \textbf{ in } t_1) \stackrel{\text{def}}{=} \text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta\{v \mapsto \text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(t)\})(t_1)$.

7. For attribute access expressions and navigation expressions, we define:

   a) $\text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(t.n) \stackrel{\text{def}}{=} \xi(\text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(t))(n)$ if t evaluates to an object and n is an attribute name, otherwise:

   b) $\text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(t.n) \stackrel{\text{def}}{=} \wr \xi(o)(n) \mid o \in \text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(t) \int$, if t evaluates to a bag of objects and n is an attribute name, otherwise:

   c) $\text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(t.n) \stackrel{\text{def}}{=} [\![self.n]\!](\text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(t))$, if t evaluates to an object and n is an association end name, otherwise:

   d) $\text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(t.n) \stackrel{\text{def}}{=} \wr \wr o' \mid \exists r : \exists f : f \in [\![r]\!] \wedge \exists n' : f(n') = o \wedge f(n) = o' \int \mid o \in \text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(t) \int$, if t evaluates to a bag of objects and n is an association end name, otherwise:

   e) the result is $\mathcal{J}$.

   For sets and sequences define the evaluation function analogously to Item 7b, respectively, Item 7d.

8. For previous attribute access expressions and previous navigation expressions, we define:

   a) $\text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(t.n@\textbf{pre}) \stackrel{\text{def}}{=} \overleftarrow{\xi}(\text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(t))(n)$ if t evaluates to an object and n is an attribute name, otherwise:

   b) $\text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(t.n@\textbf{pre}) \stackrel{\text{def}}{=} \wr \overleftarrow{\xi}(o)(n) \mid o \in \text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(t) \int$, if t evaluates to a bag of objects and n is an attribute name, otherwise:

   c) $\text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(t.n@\textbf{pre}) \stackrel{\text{def}}{=} \overleftarrow{[\![self.n]\!]}(\text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(t.))$, if t evaluates to an object and n is an association end name, otherwise:

   d) $\text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(t.n@\textbf{pre}) \stackrel{\text{def}}{=} \wr \wr o' \mid \exists r : \exists f : f \in \overleftarrow{[\![r]\!]} \wedge \exists n' : f(n') = o \wedge f(n) = o' \int \mid o \in \text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(t) \int$, if t evaluates to a bag of objects and n is an association end name, otherwise:

   e) the result is $\mathcal{J}$.

   For sets and sequences define the evaluation function analogously to Item 8b, respectively, Item 8d.

   Observe that we use $\overleftarrow{\sigma}$ instead of σ as the object diagram for obtaining the value of the attribute or navigation, which is the only difference to Item 7.

9. For method call expressions, define:

   a) If $v = \text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(t)$ is an object and $v_i = \text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(t_i)$ for all $1 \leq i \leq n$, then $\text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(t.m(t_1, t_2, \ldots, t_n)) \stackrel{\text{def}}{=} \mathcal{I}(v)(\sigma)(m)(v_1, v_2, \ldots, v_n)$, and otherwise:

b) If $v = \text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(t)$ is a bag of objects and $v_i = \text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(t_i)$ for all $1 \leq i \leq n$, then $\text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(t.m(t_1, t_2, \ldots, t_n)) \stackrel{\text{def}}{=} \wr x \mid o \in v \wedge x = \mathcal{I}(o)(\sigma)(m)(v_1, v_2, \ldots, v_n) \int$.

For sets and sequences define the evaluation function analogously to Item 9b.

Recall, that $\mathcal{I}(o)(\sigma)(n)$ is always defined, but the value returned by this function may be undefined, which also models divergence. Also, the assumption that the expression is well-typed implies that $\mathcal{I}(o)(\sigma)(n)$ is never undefined.

10. Define $\text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(t \rightarrow m(t_1, t_2, \ldots, t_n)) \stackrel{\text{def}}{=} \mathcal{I}(v)(\sigma)(m)(v_1, v_2, \ldots, v_n)$, where $v = \text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(t)$ and $v_i = \text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(t_i)$, for $1 \leq i \leq n$ for collection call expressions.

11. For iterate expressions, define:

    a) $\text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(t \rightarrow \mathit{iterate}(v_1 : T_1; a = t_a \mid t_0) \stackrel{\text{def}}{=} \text{iterate}(v_t, v_a, f)$, where

    $$\begin{aligned} v_t &= \text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(t) \\ v_a &= \text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(t_a) \\ f &= \lambda v_x.\lambda v_a.\, \text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta\{v_1 \mapsto v_x, a \mapsto v_a\})(t_0) \quad , \end{aligned}$$

    otherwise:

    b) For more than one iteration variable, define $t \rightarrow \mathit{iterate}(v_1 : T_1, \ldots, v_n : T_n; a = t_a \mid t_0) \stackrel{\text{def}}{=} t \rightarrow \mathit{iterate}(v_1 : T_1, \ldots, v_{n-1} : T_{n-1}; a = t_a \mid t \rightarrow \mathit{iterate}(v_n : T_n; a = t_a \mid t_0))$, that is, we recursively transform an iterate expression with many iterator variables into a nested iterate expression, where each iterator expression only iterates over one iterator variable.

If t evaluates to a sequence or set, then the rule under Item 11a has to be adapted, accordingly.

Recall, that iterate results in $\mathcal{J}_{\mathfrak{D}(\tau)}$, where τ is the type of t, if the iteration is unbounded.

12. For flatten expressions $t \rightarrow \mathit{flatten}()$ let $v = \text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(t)$. Then:

$$\text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(t \rightarrow \mathit{flatten}()) \stackrel{\text{def}}{=} \text{flatten}(v) \quad ,$$

if v denotes a collection, that is, either a set, a bag, or a sequence. The function flatten(v) is the union over all elements of a set, if v is a set, the bag-union over all elements of a bag, if v is a bag, and sequence concatenation of all members of a sequence, if v is a sequence. Otherwise, $\text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(t \rightarrow \mathit{flatten}()) \stackrel{\text{def}}{=} v$.

13. Define $\text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(T.allInstances()) \stackrel{\text{def}}{=} \{o \mid \tau(o) = T\}$ for all instances of T expressions and $\text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(T.allInstances@\textbf{pre}()) \stackrel{\text{def}}{=} \{o \mid \overleftarrow{\tau}(o) = T\}$ for all previous instances of T. ◊

The semantics for dereferencing attributes or navigating objects, as described in Item 7 is not straight-forward and not very consistent. For navigation expressions this semantics is not flattening, as expressed by the third case. Accessing an attribute of a collection occurs by some kind of map function, that is, if the source (t) represents a collection, then $t.a$ represents the collection of attribute values. Only if t is an object, $t.a$ results in a value. The interpretation of such an expression is, indeed, dependent on the semantic value of the source.

The intuition of the interpretation of an iterate expression, as described in Item 11 is, that the expression t_0, in which both a and the iteration variable v_1 may occur, is evaluated for each element of the collection described by t. The result of the interpretation of t_0 then becomes the new value of a and a next element is selected.

We observe that the abstract machine we have defined is similar to an abstract machine interpreting a typed functional programming language. This implies that the evaluation of a constraint may diverge because of recursive definitions. If the evaluation diverges for an expression, its semantic value is undefined, written $\mathcal{J}$.

The most notable difference between OCL and a functional programming language is, that in operation call expressions the operation to be called is bound late, that is, the actual function evaluated depends on the *type* of the callee. The motivation of this interpretation was, that OCL is supposed to be usable for programmers of object-oriented programming languages, and, therefore, should behave like one.

Definition 2.20 (Semantics of Context)**.** Let D be a class diagram and $\sigma = \langle \tau, \xi, [\![\cdot]\!] \rangle \in \Sigma$ and $\overleftarrow{\sigma} = \langle \overleftarrow{\tau}, \overleftarrow{\xi}, \overleftarrow{[\![\cdot]\!]} \rangle \in \Sigma$ be two object diagrams, such that $\sigma \models D$ and $\overleftarrow{\sigma} \models D$. Then the semantics of a constraint, that is, an OCL expression of type boolean in D, in a context is defined by:

1. For any classifier c define:

$$\text{eval}(D, \epsilon, \sigma, \eta)(\textbf{context}\ c\ \textbf{inv} : t) \stackrel{\text{def}}{=}$$
$$\forall o \in \mathbb{O} : \tau(o) = c \implies \text{eval}(D, \sigma, \epsilon, \eta\{self \mapsto o\})(t)$$

2. For any operation context $c :: m(\vec{v})$ and any action $\langle \overleftarrow{\sigma}, \sigma \rangle$, which corresponds to a call of m, such that $\vec{v} \in \text{dom}(\eta)$, define:

$$\text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(\textbf{context}\ c :: m(\vec{v})\ \textbf{pre} : t\ \textbf{post} : t') \stackrel{\text{def}}{=}$$
$$\forall o \in \mathbb{O} : \tau(o) = c \implies (\text{eval}(D, \epsilon, \overleftarrow{\sigma}, \eta\{self \mapsto o\})(t)$$
$$\wedge\ E(\xi(o)(result)) \implies \text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta\{self \mapsto o\})(t')).$$

◊

A class invariant of class c is true in an object diagram, if it holds for all instances of c. Note that a class invariant is not required to hold for instances of the subclasses of c. This implies that class invariants are not inherited. Also observe that while the evaluation function accepts a valuation of local variables η as a parameter, no local variable (with the exception of *self*) may occur freely in the invariant. Therefore, the result of eval is independent from the choice of η.

A constraint on an operation holds, if, for all actions $\langle \overleftarrow{\sigma}, \sigma \rangle$ it holds for all classes c which have executed that operations, which is similar to the interpretation of postconditions in VDM (see Jones [75]). The successful termination of an operation is indicated by $E(\xi(o)(\mathit{result}))$, where *result* is treated as a special attribute in the object diagram. It must not be an attribute for any class in the class diagram, because otherwise Definition 2.14 would be violated. Therefore, we require $\sigma \models D$, which implies that *result* is always undefined in the object diagram. On the other hand, a variable is needed in the system's state, where the result of the operation is stored, as suggested by de Roever and Engelhardt [46, p. 314f.]. To simplify the formalisation of class diagrams, object diagrams, and the relation between them, we specifically exclude *result* from the attribute and introduce it as a *logical attribute*, which is undefined in the state the operation is invoked and carries the value of the operation's result when the operation has terminated. Also, the actual parameters supplied to an operation are treated like logical attributes for the same reason. Additionally, the parameters of an operation are *immutable* during the execution of an operation, that is, they are like logical variables in Hoare logic.

Observe that the reason for choosing only objects that exist in the post-state to check whether a pre-post specification holds, are two-fold. If an object is newly created, it does not yet exist in the state of the pre-condition. The constructor method may, however, accept parameters. If the conjunction would range over the objects which don't exist in the pre-state, then the constructor methods would not be constrained. Secondly, objects may also be destroyed. Their identities do not occur in the post-state, hence no constraint is imposed on destructor methods. Indeed, the post condition can only be true, because the object does not exist anymore, once the method terminates. Because we do not allow explicit invocation of the destructor method, it is sufficient to assume that the pre-condition is true, too.

If there are an infinite number of objects defined, then in OCL all constraints are evaluated to undefined. Because OCL only expresses safety properties, it is sufficient to consider all finite prefixes of a computation, where, if the system starts with a finite number of objects in the initial state, only a finite number of objects exist, because with every step of the system, only one action is performed, which implies that at most one object can be created with each action. Therefore, the quantification over objects of a class is always a quantification over a finite set of objects.

In UML it is a semantic variation point what the meaning of a violated constraint is. One may choose to assign an operational meaning to constraints, like in Eiffel [97]. There, the violation of a behavioural specification causes the system to raise an excep-

tion, which may be handled by the program.

Since we are interested in the *correctness* of the model, a violated constraint just means, that the system is incorrect, or "ill-defined" in UML's terms.

2.3.4 OCL Standard Library

In the preceding section we have defined an interpreter for a core of OCL. This core of OCL is sufficient to encode many primitives needed for specifying systems. On the other hand, using this core is inconvenient. In this section we describe the OCL standard library, which defines commonly used types (or classes) and their operations. We define the signature of a subset of these operations together with their semantics.

We only describe a subset of the standard library. For a complete description of the standard library we refer to the OCL standard [113, Chapter 11].

Types and its Operation

Some expressions in OCL do not specify values but types. We usually write T in the signature instead of $x : T$ to indicate that this parameter is expected to be a type. Formally, we use the mechanisms for parametric classes, which are *not* used in the OCL specification.

Each expression evaluating to a type specifies one operation:

- $allInstances() : Set(T)$, where T is the type of the "callee". This operation returns all instances of the type described by the callee which exist in the current state. The OCL standard specifies that this operation may only be used if the result is a *finite* set. $Integer.allInstances()$ does not result in a finite set, but if the callee describes a class, it does.

OclInvalid, the Bottom Type

OclInvalid is a type that is a subtype of any other type. Since a type hierarchy is a partially ordered set, the types *OclInvalid* and *OclAny* (defined below) complete the subset of ground types to a complete lattice. Instances of this type represent exceptional situations like division by zero or divergence of the interpreter.

When any operation is invoked on instances of *OclInvalid* the result will be an instance of type *OclInvalid*. The only exception are *oclIsInvalid*() and *oclIsUndefined*(), which return true when invoked on instances of *OclInvalid*.

OclVoid, the Empty Type

The type *OclVoid* is a type which the standard requires to be a subtype of any other type.

The standard introduces many complications with *OclInvalid* and its literal, *OclVoid* and its literal null, and actually requiring that null is an instance of *OclVoid*. For the purposes of this thesis we require that *OclVoid* is the uninhabited type, of which no object or value is an instance. The role of *OclVoid* coincides with the unit type found in functional programming languages [126]. Operations cannot accept arguments of type *OclVoid*, but they may return *OclVoid*, which causes the interpreter of OCL to return a "token" notifying the caller that the operation has been completed. The choice of this token is arbitrary and is not observable in OCL, except for the fact that the caller continues evaluating its operations.

As a consequence, null, which represents the empty or missing object, becomes an instance of every type. Invoking any operation on null results in a value of type *OclInvalid*, or a "dereferencing null error". The only exception to this rule is the equality operator =, which returns true, if both operands are null, and false otherwise.

Observe that this definition of *OclVoid* and null is different from the definition in the OCL standard [113, p. 153]. Our definition is identical to the one used in virtually all (object-oriented) programming languages [56, 146, 57]. This alternative definition vastly simplifies the semantics of OCL and also makes it more usable (see Section 2.3.5).

OclAny, the Top Type

OclAny is the implicit super-type of all types occurring in the model, the UML specification, and the OCL standard library. An exception are parameterised types (compare to Section 2.1.3), which include OCL's collection types. The type *OclAny* defines the following operations:

- $= (x : \mathit{OclAny}) : \mathit{Boolean}$ represents equality between objects. It evaluates to true if *self* is the same as x. A more precise characterisation of equality is given in Section 2.3.5.
- $<> (x : \mathit{OclAny}) : \mathit{Boolean} \stackrel{\text{def}}{=} \mathit{not}\ a = b$ represents inequality between objects.
- $\mathit{oclIsInvalid}() : \mathit{Boolean}$ is evaluated to true whenever the callee is an instance of *OclInvalid*, and otherwise to false.
- $\mathit{oclIsUndefined}() : \mathit{Boolean}$ evaluates to true, if the callee is undefined. In the standard, this means that the callee is either equal to null or is an instance of *OclInvalid*.
- $\mathit{oclIsNew}() : \mathit{Boolean} \stackrel{\text{def}}{=} \mathit{self}@\mathbf{pre}.\mathit{oclIsUndefined}()$ can only be used in a postcondition and states that the object has been newly created during the execution of an operation.
- $\mathit{oclAsType}(T) : T$ is a "cast" or "retyping" expression, evaluating to the value of the callee, if it is an instance of T, and an instance of *OclInvalid* otherwise.

- *oclIsTypeOf*(T) : *Boolean* evaluates to true if the callee is an instance of type T.
- *oclIsKindOf*(T) : *Boolean* evaluates to true if the callee is an instance of type T or one of Ts subtypes, that is, the callee conforms to the type T (see Section 3.2).
- *oclInState*(S) : *Boolean* evaluates to true if the callee is in the state S, where S specifies a state of a state machine (see Section 2.4).

OclMessage

OclMessage is a parameterised type where its type parameter T ranges over operation and signal specifications. Instantiating the type parameter results in a concrete type usable in specifications.

OclMessage defines the following operations:

- *hasReturned*() : *Boolean* returns true if the type parameter T is an operation specification and the called operation has returned a value. In all other cases, this operation returns false.
- *result*() : S returns the value returned by an operation, if the operation has already returned a value, and an instance of *OclInvalid* in any other case. The return type S refers to the return type of the operation specification T.
- *isSignalSent*() : *Boolean* is true if T is a signal specification and false otherwise.
- *isOperationCall*() : *Boolean* is true if T is an operation specification and false otherwise.

For a precise semantics of these constructs we refer to Section 5.3.3.

Boolean and Kleene Logic

The standard type *Boolean* represents the truth values. It defines the standard connectives as operations, which accept two instances of *Boolean*, except for *not*, which only accepts one. These operations may be written using infix notation instead of the operation call notation.

The semantics of these operations is based on the one of Kleene's logic [77]. Table 2.2 displays the truth table for interpreting these operations.

In OCL logical equivalence is written using =. For clarity, we write *iff* for logical equivalence in this thesis. The reasons for this are that

1. The semantics of OCL treats boolean connectives special. They are not necessarily strict in their arguments. For example, using a strict interpretation, the OCL expression false *implies a* should be $\bot$ if $a = \bot$, similarly for *a implies* true.

a	*b*	*not b*	*a and b*	*a or b*	*a implies b*	*a iff b*	*a xor b*
true	true	false	true	true	true	true	false
true	false	true	false	true	false	false	true
true	⊥	⊥	⊥	true	⊥	⊥	⊥
false	true	false	false	true	true	false	true
false	false	true	false	false	true	true	false
false	⊥	⊥	false	⊥	true	⊥	⊥
⊥	true	false	⊥	true	true	⊥	⊥
⊥	false	true	false	⊥	⊥	⊥	⊥
⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥

Table 2.2: Semantics of boolean connectives

2. The semantics of = is not precisely defined for all types occurring in a model by the standard, but for *Boolean* we have a precise definition for =.

3. It removes an inconsistency, because OCL introduces *xor*, which is semantically equivalent to <>.

Real

The standard data type *Real* represents real-valued quantities. This class defines the usual mathematical connectives with their standard semantics. We enumerate these connectives in order to provide their signature, which is used for Chapter 3.

- $+(x : Real) : Real$ describes the sum of *self* and x.
- $-(x : Real) : Real$ describes the difference between *self* and x.
- $*(x : Real) : Real$ describes the product of *self* and x.
- $/(x : Real) : Real$ describes the quotient of *self* and x.
- $-() : Real \stackrel{\text{def}}{=} 0 - self$ describes the negation of *self*.
- $< (x : Real) : Boolean$ evaluates to true, if the value of *self* is less than the value of x. It evaluates to false, if the value of *self* is equal to or greater than the value of x. In any other case, it is undefined.
- $> (x : Real) : Boolean$ evaluates to true, if the value of *self* is greater than the value of x. It evaluates to false, if the value of *self* is less than or equal to the value of x. In any other case, it is undefined.

- $<=(x : Real) : Boolean$ evaluates to true, if the value of *self* is less than or equal to the value of x. It evaluates to false, if the value of *self* is greater than the value of x. In any other case, it is undefined.
- $>=(x : Real) : Boolean$ evaluates to true, if the value of *self* is equal to or greater than the value of x. It evaluates to false, if the value of *self* is less than the value of x. In any other case, it is undefined.
- $abs() : Real \stackrel{def}{=} \textbf{if } self < 0 \textbf{ then } -self \textbf{ else } self \textbf{ endif}$ describes the absolute value of *self*.
- $floor() : Integer$ describes the largest integer which is not greater than *self*.
- $round() : Integer \stackrel{def}{=} (self + 0.5).floor()$ rounds *self* to the nearest integer.
- $max(x : Real) : Real \stackrel{def}{=} \textbf{if } self < x \textbf{ then } x \textbf{ else } self \textbf{ endif}$ results in the greater value of *self* and x.
- $min(x : Real) : Real \stackrel{def}{=} \textbf{if } self > x \textbf{ then } x \textbf{ else } self \textbf{ endif}$ results in the smaller value of *self* and x.

Integer

The standard data type *Integer* represents integer-valued quantities and is a sub-class of *Real*. This class defines the usual mathematical connectives with their standard semantics. We enumerate these connectives in order to provide their signature, which is used for Chapter 3.

- $+(x : Integer) : Integer$ describes the sum of *self* and x.
- $-(x : Integer) : Integer$ describes the difference between *self*
- $*(x : Integer) : Integer$ describes the product of *self* and x.
- $/(x : Integer) : Real$ describes the quotient of *self* and x.
- $-() : Integer \stackrel{def}{=} 0 - self$ describes the negation of *self*.
- $<(x : Integer) : Boolean$ evaluates to true, if the value of *self* is less than the value of x. It evaluates to false, if the value of *self* is equal to or greater than the value of x. In any other case, it is undefined.
- $>(x : Integer) : Boolean$ evaluates to true, if the value of *self* is greater than the value of x. It evaluates to false, if the value of *self* is less than or equal to the value of x. In any other case, it is undefined.

- $<=$ (x : *Integer*) : *Boolean* evaluates to true, if the value of *self* is less than or equal to the value of x. It evaluates to false, if the value of *self* is greater than the value of x. In any other case, it is undefined.

- $>=$ (x : *Integer*) : *Boolean* evaluates to true, if the value of *self* is equal to or greater than the value of x. It evaluates to false, if the value of *self* is less than the value of x. In any other case, it is undefined.

- *abs*() : *Integer* $\stackrel{\text{def}}{=}$ **if** *self* < 0 **then** $-$ *self* **else** *self* **endif** describes the absolute value of *self*.

- *div*(x : *Integer*) : *Integer* describes Euclidean division of *self* by x, that is, it results in the unique integer y such that there exists an $0 \leq r < x$ with $y * x + r =$ *self*.

- *mod*(x : *Integer*) : *Integer* describes Euclidean reminder of *self* divided by x, that is, it results in the unique integer $0 \leq r < x$ such that there exists an integer y with $y * x + r =$ *self*.

- *max*(x : *Integer*) : *Integer* $\stackrel{\text{def}}{=}$ **if** *self* $< x$ **then** x **else** *self* **endif** evaluates to the greater value of *self* and x.

- *min*(x : *Integer*) : *Integer* $\stackrel{\text{def}}{=}$ **if** *self* $> x$ **then** x **else** *self* **endif** evaluates to the smaller value of *self* and x.

Observe that the arithmetic operators have been overloaded to assert that the result is an integer. Overloading / is redundant, because the version defined in *Real* is the same operator.

String, Unlimited Integer

The string data type and the unlimited integer data type are not used in this thesis. We refer to the OCL standard [113, Section 11.5.3 and Section 11.4.5] for a definition of these data types and their semantics.

Collection

The type *Collection* (and its subtypes) is a parameterised type[10], with the type parameter T specifying the type of the elements. A concrete type can be obtained by instantiating the template parameter. For example, *Set* is a parameterised type (which cannot be instantiated), while *Set*(*Integer*) is; it is the type of a set of integers.

[10] See also Section 2.1.3 for parameterised classes and types.

Most operations on collections can be expressed using iterate expressions or iterator expressions. These expressions are defined in Definition 2.19, Item 11, and in the section on *predefined iterator expressions* below (pp. 49ff.)

The following operations are defined on *Collection*:

- $size() : Integer \stackrel{def}{=} self \rightarrow iterate(e; a = 0 \mid a + 1)$
- $count(e : T) : Integer \stackrel{def}{=} self \rightarrow iterate(i; a = 0 \mid \textbf{if } e = i \textbf{ then } a + 1 \textbf{ else } a \textbf{ endif})$ counts how often e occurs in the collection.
- $includes(e : T) : Boolean \stackrel{def}{=} self \rightarrow count(e) > 0$ returns true if and only if e occurs in the collection.
- $excludes(e : T) : Boolean \stackrel{def}{=} self \rightarrow count(e) = 0$ returns true if and only if e does not occur in the collection.
- $includesAll(c : Collection(T)) : Boolean \stackrel{def}{=} c \rightarrow forAll(e \mid self \rightarrow includes(e))$.
- $excludesAll(c : Collection(T)) : Boolean \stackrel{def}{=} c \rightarrow forAll(e \mid self \rightarrow excludes(e))$.
- $isEmpty() : Boolean \stackrel{def}{=} self \rightarrow size() = 0$.
- $notEmpty() : Boolean \stackrel{def}{=} self \rightarrow size() <> 0$.
- $sum() : T \stackrel{def}{=} self \rightarrow iterate(e; a = 0 \mid a + e)$. It is not possible to formally define this operation using the means provided by OCL, a formal and precise definition is given in Section 3.3.4.

Set

Set is the type of mathematical sets.

- $= (c : Set(T)) : Boolean$ describes set-equality.
- $including(e : T) : Set(T)$ describes the set obtained from *self* by including e.
- $excluding(e : T) : Set(T)$ describes the set obtained from *self* by excluding e.
- $union(c : Set(T)) : Set(T)$ describes the union of the set *self* and the set c.
- $union(c : Bag(T)) : Bag(T)$ describes the union of the bag obtained from *self* by assuming that each element of *self* occurs exactly once and the bag c.
- $intersection(c : Set(T)) : Set(T)$ describes the intersection of the set *self* the set c.

- *intersection*(c : *Bag*(T)) : *Set*(T) describes the intersection of the set *self* and the set obtained from c by including every element contained in c.
- $-(c$: *Set*(T)) : *Set*(T) describes the difference of the set *self* the set c.
- *flatten*() : *Set*(T'). If *self* is a set of collections, then this operation returns the set-union of all its elements. Otherwise the result is the value of *self*. It is not possible to formally define this operation using the means provided by OCL; a formal and precise definition is given in Definition 2.19, Item 12, and in Section 3.3.5.
- *asSet*() : *Set*(T) $\stackrel{\text{def}}{=}$ *self* $\rightarrow$ *iterate*(e; a = *Set*{} | $a \rightarrow$ *including*(e)) is the identity function and only defined for symmetry reasons.
- *asBag*() : *Bag*(T) $\stackrel{\text{def}}{=}$ *self* $\rightarrow$ *iterate*(e; a = *Bag*{} | $a \rightarrow$ *including*(e)) returns a bag which includes each element of *self* exactly once.
- *asSequence*() : *Sequence*(T) returns a sequence containing all elements of *self* exactly once. The order of the elements is arbitrary. It is equivalent to the expression

$$self \rightarrow iterate(e; a = Sequence\{\} \mid a \rightarrow append(e)) \quad .$$

Bag

Bag is the type of a bag or multi-set. Bags usually arise as the result of navigation expressions.

- $=$ (c : *Bag*(T)) : *Boolean* describes equality of multi-sets.
- *including*(e : T) : *Bag*(T) describes the bag obtained from *self* by including e.
- *excluding*(e : T) : *Bag*(T) describes the bag obtained from *self* by excluding all occurrences of e.
- *union*(c : *Bag*(T)) : *Bag*(T) describes the union of the bag *self* and the bag c.
- *union*(c : *Set*(T)) : *Bag*(T) describes the union of the bag *self* and the bag obtained from c by including each element of c exactly once.
- *intersection*(c : *Set*(T)) : *Set*(T) describes the intersection of the bag *self* and the set c.
- *intersection*(c : *Bag*(T)) : *Bag*(T) describes the intersection of the bag *self* and the bag c.

- $-(c : Set(T)) : Set(T)$ describes the bag obtained from *self* by removing all occurrences of the elements of c.
- *flatten*() : $Bag(T')$. If *self* is a bag of collections, then this operation returns the bag.union of all its elements. Otherwise the result is the value of *self*. It is not possible to formally define this operation using the means provided by OCL; a formal and precise definition is given in Definition 2.19, Item 12, and in Section 3.3.5.
- $asSet() : Set(T) \stackrel{\text{def}}{=} self \rightarrow iterate(e; a = Set\{\} \mid a \rightarrow including(e))$ returns a set which contains each element of *self*.
- $asBag() : Bag(T)$ is the identity function and only defined for symmetry reasons.
- $asSequence() : Sequence(T)$ returns a sequence containing all elements of *self* as often as they occur in the multi-set. The order of the elements is arbitrary. It is equivalent to:

$$self \rightarrow iterate(e; a = Sequence\{\} \mid a \rightarrow append(e)) \quad .$$

Sequence

Sequence is the type of mathematical sequences.

- $at(i : Integer) : T$ results in the element at the ith position of the sequence.
- $= (c : Sequence(T)) : Boolean \stackrel{\text{def}}{=}$ **let** $s = self \rightarrow size()$ **in** $s = c \rightarrow size()$ *and* $Sequence\{1..s\} \rightarrow forAll(i : Integer \mid self \rightarrow at(i) = c \rightarrow at(i))$ describes equality of sequences.
- $union(c : Sequence(T)) : Sequence(T)$ describes the concatenation of *self* and c.
- *flatten*() : $Sequence(T')$. If *self* is a sequence of collections, then this operation returns the sequence concatenation of all its elements. Otherwise the result is the value of *self*. It is not possible to formally define this operation using the means provided by OCL; a formal and precise definition is given in Definition 2.19, Item 12, and in Section 3.3.5.
- $append(e : T) : Sequence(T) \stackrel{\text{def}}{=} self \rightarrow union(Sequence\{e\})$ results in the sequence which consists of all elements of *self* with e appended.
- $prepend(e : T) : Sequence(T) \stackrel{\text{def}}{=} Sequence\{e\} \rightarrow union(self)$ results in the sequence which consists of all elements of *self* with e prepended.
- $excluding(e : T) : Sequence(T) \stackrel{\text{def}}{=} self \rightarrow iterate(i; a = Sequence\{\} \mid$ **if** $e = i$ **then** a **else** $a \rightarrow append(i)$ **endif**) results in the largest sub-sequence of *self*, in which e does not occur.

- $subSequence(l : Integer, u : Integer) : Sequence(T)$ results in the subsequence of *self* starting at index l and ending at index u. It is equivalent to:

$$Sequence\{1..u\} \rightarrow iterate(i; a = Sequence\{\} \mid \textbf{if } l \leq i \textit{ and } i \leq u \textbf{ then } a \rightarrow append(self \rightarrow at(i)))$$

- $insertAt(e : T, i : Integer) : Sequence(T) \stackrel{\text{def}}{=} (self \rightarrow subSequence(1, i-1) \rightarrow append(e)) \rightarrow union(self \rightarrow subSequence(i, self \rightarrow size()))$ results in a sequence where e is inserted at position i.

- $first() : T \stackrel{\text{def}}{=} self \rightarrow at(1)$.

- $last() : T \stackrel{\text{def}}{=} self \rightarrow at(self \rightarrow size())$.

- $asSet() : Set(T) \stackrel{\text{def}}{=} self \rightarrow iterate(e; a = Set\{\} \mid a \rightarrow including(e))$ returns a set which contains each element of *self*.

- $asBag() : Bag(T) \stackrel{\text{def}}{=} self \rightarrow iterate(e; a = Bag\{\} \mid a \rightarrow including(e))$ returns a bag containing all elements of the sequence *self*.

- $asSequence() : Sequence(T)$ is the identity function and only defined for symmetry reasons.

Ordered Set

OrderedSet is the type of an ordered set, that is, a set with a linear order defined over its elements. Because we do not use this type in this thesis, we refer to [113, Sect. 1.7.3] for a description of the operations defined for this type and their semantics.

Predefined Iterator Expressions

Iterator expressions are not operations. However, the pre-defined iterator expressions are treated like operations of collection types. This section lists all predefined iterator types.

Quantification Quantification in OCL is expressed as an iterate expression. Indeed, the quantified expressions are defined in term of iterations:

1. $t \rightarrow forAll(v \mid t') \stackrel{\text{def}}{=} t \rightarrow iterate(v; a = \text{true} \mid a \textit{ and } t')$

2. $t \rightarrow exists(v \mid t') \stackrel{\text{def}}{=} t \rightarrow iterate(v; a = \text{false} \mid a \textit{ or } t')$

These iterate expressions describe the expansion of the quantifier into a conjunction or disjunction of the predicate t' for each element of the collection described by t.

Similar to the general iterate expression, a quantified expression may have more than one iterator variable. In this case the quantification is expanded analogously to Item 11b of Definition 2.19.

The expansion of quantifiers to iterators implies that if t represents an infinite set of values, then the meaning of the expression is undefined, because the evaluation of the iterate expression will not terminate. As an example and a consequence of the semantics of OCL, the expression $t \rightarrow \mathit{forAll}(v \mid \text{true}).\mathit{oclIsUndefined}()$ is testing the finiteness of t in OCL, whereas in traditional logic, where $\forall v : \text{true}$ is a tautology, and therefore this expression is always defined.

2.3.5 Critique of OCL

In the preceding section we have discussed the syntax and the semantics of OCL and its standard library, as described in the OCL standard and related publications. In this section we discuss the semantic decisions of the authors of the OCL standard and argue why it is necessary to define a semantics that is different from the one described in the standard. Such a semantics will be given in Section 4.5.3.

When OCL was designed, Warmer and Kleppe formulated the following requirements for OCL [154]:

> 1. OCL must be able to express extra (necessary) information on the models and other artifacts used in object-oriented development.
> 2. OCL must be a precise, unambiguous language that can easily be read and written by all practitioners of object technology and by their customers. This means that the language must be understood by people who are not mathematicians or computer scientists.
> 3. OCL must be a declarative language. Its expressions can have no side-effects; that is, the state of a system must not change because of [evaluating] an OCL expression. [...]
> 4. OCL must be a typed language. [...]

Expressiveness

If we suppose that all "artifacts" we are concerned with are present in the UML diagram, then OCL is indeed able to express extra information on the model, and as a consequence Requirement 1 is satisfied. Whether all extra information can be specified is not clear at all, because the meaning of "extra (necessary) information" has deliberately been unspecified.

Indeed, the way we have formalised the standard library demonstrates that most concepts of that library can be encoded in a relatively small core. Only a few expressions have to be treated specially, mostly because of the object-oriented style.

Precise and Unambiguous

OCL fails on Requirement 2. In the following paragraphs we give a variety of reasons, of which most of them are technical. To summarize these reasons: OCL is a complex language with a complex semantics, which is only understandable by experts in the field.

Syntax It is hard to believe that OCL "can easily be read [...] by [...] customers", that is, not by mathematicians and computer scientists. The syntax of OCL follows conventions of object-oriented programming languages, for example using a colon and an arrow for "sending a message", and by this implying a message passing paradigm even in the specification language. Especially that quantification and iteration are written using a post-fix notation, whereas many operators of the standard data types are written in a more conventional infix notation, makes OCL very hard to read. The recent proposal by Jos Warmer of using an SQL-like syntax, which is much closer to natural languages, has, however, not yet found many supporters.

Semantics OCL also has to be understood by its users. In order to satisfy this requirement, one needs a precise and *simple* semantics which does not include surprises. But until now no consensus has been reached on how to interpret a specification in OCL among *computer scientists* and language designers, as the literature (see Section 2.5 for a selection of references) and the "OCL Issue List" [112] demonstrate.

Late Binding and Open Recursion Late binding (often called *dynamic dispatch* or *virtual binding*) and open recursion are the fundamental properties of object-oriented programming. By late binding we mean that the run-time type of the object decides how an operation call is to be interpreted. Open recursion means that a recursive call, here, the call of any operation on the object represented by *self*, is bound late.

Because OCL allows to call any side-effect free operation defined in a UML model, late binding has to be introduced into the semantics of OCL. Therefore, each OCL constraint is subject to the effect of the *fragile base class problem*. This means, that changes in a subclass can change the behaviour defined in the super-class in unexpected ways.

If late binding is used disciplined, either by avoiding *overriding*, that is, redefining the implementation of an operation, or *overloading*, that is, defining operations with the same name, but with a different signature, or by asserting that overriding and overloading coincide with behavioural subtyping, reasoning with OCL is not affected.

But in most models overriding is used in an ad-hoc fashion. The implementation of an operation is overridden with a new implementation which behaves completely different. For most modellers, adhering to Liskov's substitution principle [94], that any instance of a sub-type can be used in place of an instance of its super-type, is less important than code-reuse.

In effect, the result of this semantics is, that in order to understand the meaning of a constraint, one has to understand the complete model and how it has been implemented. But we desire a formalism that abstracts from these details of the implementation.

Three-Valued Logic OCL has a three-valued logic. In itself a three-valued logic does not add much complexity to a specification language. The reasons for this are that each partial function f is extended in the semantics of OCL to a *strict* total function $f_\perp \stackrel{\text{def}}{=} \lambda x.\text{if } x \in \text{dom}(f) \text{ then } f(x) \text{ else } \perp \text{ fi}$ and that relations and predicates are interpreted as strict functions to the three truth-values.

The undefined value also arises from the (implicit) requirement that each constraint should represent a *computable* function (see below how this requirement affects Requirement 3). But it is important to note what the additional truth value, undefined, is supposed to represent.

Usually, a formula whose meaning is undefined is considered to be nonsensical. In OCL, this is represented by calling such formulae and the models for which they have to hold, "ill-formed". Any ill-formed model is considered to contain an error. Consequently, in most cases, a constraint that is undefined, can also be considered to be false.

In OCL, however, the undefined value is a first-class citizen. This is represented by the operation *oclIsUndefined*(), which is used to test whether a value is undefined. The interpretation $\mathcal{I}(t)(\sigma)(oclIsUndefined)$ is defined as follows:

$$\mathcal{I}(v)(\sigma)(oclIsUndefined) \stackrel{\text{def}}{=} \begin{cases} \text{true} & \text{, if } v = \perp \\ \text{false} & \text{, otherwise.} \end{cases}$$

A specification which evaluates to undefined, can be turned into a well-formed specification by applying the *oclIsUndefined* operation to it.

One central complication of the semantics of OCL is that the undefined value $\perp$ represents both the concept of "run-time error" and the concept of "divergence". Both notions should *not* be concepts in the semantics of a specification language. These notions are important in the program to reason about. A specification is not meant to be executed, but to define desired properties of the program or model.

A "run-time error" means, that an OCL expression contains a function application where the argument is outside the domain of the function. Such constraints should be regarded as nonsensical, and the constraint is then considered to be false.

"Divergence" means, that a recursive OCL expression has no fixed-point or that this fixed-point is not computable using applicative order evaluation, that is, when all ar-

guments to a function are evaluated first before the function is applied. Especially, the second case should be of no concern for the semantics of a specification language, because it is then undefined because of a particular implementation of an interpreter of the language. The meaning of quantification over infinite domains cannot be computed (or tested). However, many OCL expressions have a well-defined meaning in a declarative semantics. For Example, $T.allInstances() \rightarrow forAll(\text{true})$ should mean the same as $\forall(x : T) : \text{true}$, which is, of course, a tautology. In OCL, the modified constraint $T.allInstances() \rightarrow forAll(\text{true}).oclIsUndefined()$ tests whether the domain of the type T is finite.

Decidability The unpleasant side-effect of *oclIsUndefined* is, that it is impossible to implement a correct, that is, standard-conforming, interpreter for OCL. The proof for this statement is defined as follows:

Theorem 2.21. *There exists no computable function f such that, for all constraints t, $f(t) = \text{true}$ if and only if $\text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(t) = \text{true}$.*

Proof. Let t' be an OCL constraint such that $\text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(t')$ diverges for some D, $\overleftarrow{\sigma}$, σ, and η. Choose $t \stackrel{\text{def}}{=} t'.oclIsUndefined()$. Clearly:

$$\text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(t) = \text{true} \quad .$$

Suppose f is a computable function with $f(t'.oclIsUndefined()) = \text{true}$. Because this function is computable, it has to terminate on t', therefore solving the Halting problem. Consequently, no such computable function exists. □

A corollary of this theorem is that eval is not a standard-conforming implementation of an OCL interpreter. Indeed, if $\text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(t) = \bot$ for any t where eval is diverging, then $\text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(t.oclIsUndefined()) = \bot$.

Equality For three-valued logics introducing an undefined value, we have to distinguish two kinds of equality. The first one is strict in its arguments, that is, if one of the arguments is undefined, so is the equality expression. This equality is called *weak equality* and is written $\doteq$. The operator *iff* is indeed weak equality on *Boolean*.

The second one is called *strong* equality, written as $=$. Its semantics is not strict, but the undefined value is treated as a unique value by $=$. Consequently, the term $a = b$ is either true or false.

OCL defines equality $=$ as a property on the type *OclAny*, which can be overridden, that is, redefined, in subclasses of *OclAny*. This gives the flexibility to define equality by structure or equality by reference (or, God forbid, define $=$ to mean something completely different). The evaluation rules for property-call expressions, which $=$ is, have to be applied. This implies that equality is *weak*. For example, the constraint $1/x = 1/x$ is *not* necessarily true. If $x = 0$, then this constraint is undefined.

This violates the axiom of self-identity ($\vdash x = x$) used in virtually all deductive systems, which also represents one of the major surprises to users of OCL. The correct formulation of $1/x = 1/x$ in OCL is to write $x \neq 0$ *implies* $1/x = 1/x$, which is true for all x (see Table 2.2).

For reasoning it is preferable to use strong equality instead of weak equality, as defined in OCL. The reason for this is, that we cannot replace equals for equals anymore, which complicates reasoning a lot.

Meaning of @pre The meaning of the **@pre** modifier is confusing to users, because it does not apply to an expression as a whole, but only to the part it has been applied to. This operator allows to mix values from the pre- and the post-state of an operation call freely. This flexibility is needed, but for unexperienced users, a notation which allows to apply **@pre** to an expression as a whole is desirable. But very often, expressions are overloaded with these annotations, for example:

context $c :: m() : Integer$
post : $result = (self.a@\textbf{pre} * self.a@\textbf{pre} +$
$self.b@\textbf{pre} * self.b@\textbf{pre}).sqrt()$

Characterisation OCL defines a scientifically but complex theory, which can be characterised as a first-order relational theory without equality but with late binding and fix-points. Especially the presence of late binding in a logic leads to a theory that is not yet well understood and too complex for the original purpose it was designed for. The lack of equality is surprising for many users.

Declarative

Requirement 3 states, that OCL must be a *declarative language*, which has been characterised by the absence of side-effects. OCL satisfies this requirement.

Typed Language

Whether a specification language (or constraint language) has to be typed is a matter of preference. Virtually all specification languages are typed. All of them have *manifest types*, that is, the type of the object denoted by a variable is used by the reasoning. The expression $a + b$ only has a meaning if a and b denote numerical quantities, for example, their manifest type is *Real*. Adding a set to a number is, on the other hand, considered to be nonsensical.

The more important and interesting question is, whether a specification language should be *statically typed*. This implies that all entities in a specification have a declared type and a *type checker* proves that all formulae are well-typed with respect to a type system.

OCL is a statically typed language. However, the type system used for OCL is weak, in the sense that the most precise type of each entity cannot be determined. Even the extended type system described in Chapter 3 is weak. This is a consequence of requiring a decidable type system for a statically typed language. In the case of OCL, this implies that the type system has to lose information, that is, the types computed for expressions are actually super-types of the manifest type of the expression.

OCL solves this problem by introducing re-typing operations (or casts). The use of these additional operations force the type checking procedure to assume that the manifest type of an expression is a sub-type of the type the expression is re-typed to. However, the type checker cannot prove whether the cast is justified (otherwise it would not have been necessary in the first place), so it is possible to cast an expression into a type it is not a value of. This results in an undefined value, a situation static type-checking is supposed to avoid.

On the other hand, a sufficiently strong type system, like the one used for the PVS specification language of Higher Order Logic [121, 93], is much more expressive, avoids all casts, but is not decidable. If these systems cannot determine whether a specification is well-typed, they generate proof obligations which have to be proved interactively.

2.4 State Machines

UML offers many methods for describing the behaviour of objects. One method used for specifying the behaviour of *active* objects is using UML state machines. UML state machines have evolved from Harel's state charts [60] via the object-oriented successors of state charts [61].

In this section we summarise the notation of state machines and briefly review its semantics. In this thesis we only use *flat state machines*, that is, state machines in which no state is contained in another state. Such states are also called simple.

The basic concepts of UML 2.0 state machines are states and transitions between them. Transitions are labelled with a *trigger*, which specifies a signal, which may cause the transition to be taken, followed by a guard, which is a boolean condition on the state of the system as defined by an object diagram. The final component of a transition label is the action, which will be performed if the transition *fires*, that is, is taken to obtain a new state. This label is written $t[g]/a$, where t represents the trigger, g represents the guard, and a represents the action. Components may be omitted from the label, which also leads to the omission of the decoration, for example, if the guard is omitted, we simply write t/a, if the trigger and the guard is omitted, we write $/a$, and if the transition is labelled with only the trigger we write t as a label.

The environment may send events to the state machine. These events are collected in the event pool of the state machine. A state machine may dispatch a single event from its event pool to trigger transitions.

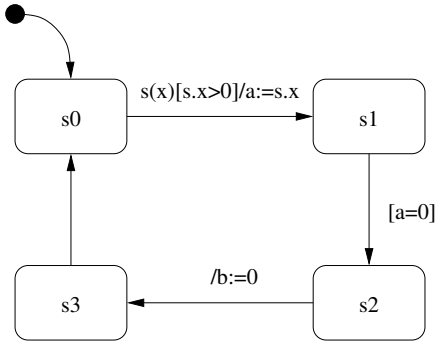


Figure 2.4: State machine

A transition is enabled if its source state is currently active, its event is dispatched from the event pool, and its guard evaluates to true. The firing of transition t leads, in this order, to:

1. the deactivation of the state that will be left by firing t,
2. the execution of the actions of t (which we simply call the execution of t)
3. the activation of its target state.

UML state machines use a *run-to-completion* semantics, that is, "an event occurrence can only be taken from the pool and dispatched if the processing of the previous current occurrence is fully completed"

We give an intuitive explanation of the syntax and semantics using the state machine displayed in Figure 2.4. The state $s0$ is the initial state of the state machine, as indicated by the transition leading to $s0$ from the black dot. While the state $s0$ is active, the next transition can only be activated by a signal of type s, which defines at least the attribute x, indicated by the trigger $s(x)$, such that the value of $s.x$ is larger by one, which is indicated by the guard $s.x > 0$. While in state $s1$, the state machine awaits that the value of its attribute a is changed to 0. In this case, the state of the state machine is changed to $s2$, and no action is performed. Such transitions are triggered by so-called *change events* which are generated whenever a guard becomes true. Note that the change event will *not* be removed from the event pool, if the value of the condition is changed to false after the event has been generated. There is only one outgoing transition leaving $s2$, which is neither guarded by a trigger nor a guard. Therefore, it is always enabled. If neither a trigger nor a guard is defined for a transition, then it is taken, whenever a so-called completion event is generated. In our case, such events are always generated once a transition has been taken. When the transition is taken, then the attribute b will be set to 0. Finally, we have an unlabelled transition from $s3$ to $s0$, which will be taken with a completion event, and where no action will be performed. Whenever an event is chosen from the event pool, which does not trigger a transition,

it is discarded, unless it is declared *deferrable*. Deferrable events are placed back into the event pool and a new event will be chosen from the event pool.

The actions defined in this thesis are:

- Assignment to attributes, written $a := e$, where a is an attribute name of the object, whose behaviour is to be described, and e is an OCL expression.
- Object creation, written $a :=$ new c, where a is an association end and c is the name of a class. If the multiplicity of the association end is 0..1 or 1, the semantics is that the new object will *replace* the object associated to the creating object. Otherwise the end refers to a collection of objects and the newly created object is *added* to the collection. If adding the newly created object violates a multiplicity constraint, the system may raise a run-time error once the system becomes stable.[11]
- Signal emission, written $a!s(\vec{e})$, where a is an association end, s is a signal name, and $\vec{e}$ is a list of expressions used to initialise the attributes of s.
- Operation calls, written $a.m(\vec{e})$, where a is an association end, m is a signal name, and $\vec{e}$ is a list of expressions providing the actual arguments to m.

The semantics of state machines used in this thesis has been provided by Hooman and van der Zwaag [149]. In their paper a formalisation of the semantics in PVS is defined, which we use in Chapter 4. Their semantics is based on the semantics described by Damm et al. in [39]. Fecher, Kyas, de Roever, and de Boer describe a compositional semantics of state machines based on the before-mentioned semantics in [52]. Because the semantics described in these papers differ considerably from the semantics described in UML 2.0 [111] state machines, we refer the reader to Schönborn [140] for a description of the semantics which is closer to the standard.

2.5 Summary

In this chapter we have introduced the essential notations of the UML 2.0 standard used in this thesis. We have defined a formal semantics in terms of an interpreter for these core concepts. For other operations of OCL the semantics is defined in terms of other OCL expressions.

The formal definition of the semantics closely follows the standards [111, 113], where we deviate from the standard if inconsistencies arise. In some cases, we simply omitted constructs which cause problems, for example, if we were not able to determine the intention or the meaning of the construct in the specification.

[11] This semantics allows, for example, initialising a collection with multiplicity $x..*$, where $x > 1$, by successively adding objects to the collection.

The approach for interpreting UML class diagrams as algebraic signatures has been described Börger et al. [14], Reggio et al. [132], Grosse-Rhode [58], and others. We follow a very similar approach.

For example, Börger et al. [14] define the dynamic behaviour of UML state machines using abstract state machines [59].

Grosse-Rhode uses many-sorted algebras to interpret class diagrams [58]. The dynamic behaviour of such a system is modelled by *transformation systems*, which are very similar to Gurevich's abstract state-machines [59]. Inheritance is modelled by signature-morphisms.

Formal semantics for OCL have been proposed by Richters [133] and by Baar [7]. Our definition of OCL 2.0 follows their definitions in many respects, but we also take the changes and new constructs of OCL 2.0 [113], for example, the null literal, into account. A second difference between our formalisation and their formalisation is, that we use higher-order logic as formal meta-language instead of a first-order predicate calculus and Zermelo-Fraenkel set-theory.

Brucker presents a deep embedding of OCL into Isabelle/HOL [101] in [19]. He proposes many improvements to the then current semantics. But he does not take the clarifications described in the "Amsterdam Manifesto" [34] into account, which includes many clarifications that where worked into the later versions of the OCL standard.

Finally, the interpretation of invariants and pre-/postconditions is very similar to the interpretation used by of Richters and Gogolla [134] and Hennicker, Hussmann, and Bidoit [62]. However, in both references it has been left unspecified how the actual arguments and the result is represented in states.

Indeed, the semantics of OCL displays problems similar to the ones observed by MacQueen [95]: If a declarative language, in MacQueens paper a functional language like ML, has to be extended with a full-fledged class-based object system, then this "leads to an excessively complex type system and relatively little expressive gain, *especially if we aim to preserve that mostly functional style of programming*" (Emphasis added by us).

The semantics of navigation expressions in OCL and UML are related to relational algebras and are inspired by relational database systems, which makes it appealing to use Jackson's Alloy Language for formalising OCL and UML [69].

A formal semantics for UML and OCL enables tool vendors to build tools which allow the exchange of models without unexpected reinterpretations by the tool, which may in practise defeat the purpose of communicating models and ideas, as well as the interchange format XMI [108]. A formal semantics also enables developers to "calculate" implementations from specifications, for example by stepwise refinement. An executable subset of UML can then be used for rapid prototyping and simulation, which helps in detecting design errors early.

Chapter 3

Type Checking OCL

We describe the design and implementation of a type-checker for OCL. Based on the experience gained in implementing and using this type checker, we observed that OCL is not suitable for constraining systems under development, because changes in the underlying class diagram unnecessarily invalidate the type correctness of constraints, whereas the semantic value of these constraints do not change. A second observation was, that the type system specified in the OCL standard does not support all language features defined for UML class diagrams and part of the OCL standard library. OCL and UML support parameterised classes (see Section 2.1.3), for example, the *Set* type of OCL, but cannot take advantage of these type abstractions, not only during type checking.

To alleviate these problems, the type system of OCL has been extended with intersection types, union types, and bounded operator abstraction. This allows to take advantage of the universal polymorphism offered by parameterised classes, and makes the well-typedness of constraints more robust during the evolution of the contextual class diagram.

3.1 Introduction

In order to be used widely, OCL needs to have a type system that is compatible with the well-formedness constraints of UML 2.0 class diagrams and that integrates with this type system seamlessly. Furthermore, the type system has to be robust with respect to model transformations, for example, refactoring[1] or other changes in class diagrams.

Influenced by our experience in implementing a type-checker for OCL, that conforms to the OCL standard, we have come to the conclusion that OCL does not adequately implement these requirements:

[1] *Refactoring*, a term initially introduced by William F. Opdyke and Ralph E. Johnson in [119], is the process of rewriting a program to improve its readability or structure, with the explicit purpose of preserving its meaning and behaviour. The main purpose of refactoring is to improve the internal consistency, remove redundancy, and often to simplify the structure. Refactoring is often automated, using a catalogue of known refactoring methods. More information on refactoring can be found in Opdyke's thesis [118] and in the book of Fowler et al [55].

The type system of OCL appears to be designed for languages which only use single inheritance and no parameterised classes[2]. UML 2.0 introduces a new model for templates, which allows classifiers to be parameterised with classifier specifications, value specifications, and operation specifications [111]. The OCL 2.0 proposal [113] does not specify how those parameters can be used in constraints, how they can be constrained, or how these parameters are to be used in constraints and what their meaning is supposed to be.

Furthermore, OCL constraints are fragile under the operations of refactoring of class diagrams, package inclusion and package merging, as explained in Section 3.3. These operations, which often have no effect on the semantics of a constraint, can render constraints ill-typed. This essentially limits the use of OCL to a-posteriori specification of class diagrams.

To solve these problems, we have implemented a more expressive type system based on intersection types, union types, and bounded operator abstraction. Such type systems are already well-understood [126] and solve the problems we encountered elegantly. Our type system supports templates and is more robust under refactoring and package merging than the current type system.

The adaption of the type system to OCL was straight forward. The specification of the OCL standard library had to be changed to make use of the new type system. We have implemented this type system in a prototype tool and all constraints of the OCL 2.0 standard library have been shown to be well-typed with respect to our type system.

This chapter is organised as follows: In Section 3.2, we survey the current type system for OCL. In Section 3.3, we describe our different extensions to the type system. In Section 3.4, we summarise the most important results. In Section 3.5 we compare our results with other results and draw some conclusions.

3.2 State of the Art

In this section, we recall the current type system used for OCL, which has been derived from the OCL 2.0 standard. It is similar to the one presented in [25].

Recall that the abstract syntax of OCL has been defined in Section 2.3.

We now define the abstract syntax of OCL types. We have essentially two kinds of types: elementary types and collection types. The elementary types are classifiers from the model and the elementary data types like *Boolean, Integer,* and so on. The collection types are types which are generic, that is, they construct a type by applying the collection type to any other type.[3] This distinction is formalised with a kinding system (a type system for types). Kinds are defined by the language $K ::= \star \mid K \to K'$. The kind $\star$ denotes any type which does not take an argument. Type constructors have a kind $K \to K'$, which means that such a constructor maps each type of kind K to a

[2]For example Java before version 5.0.

[3]In Section 3.4 we discuss the presence of *dependent types* in class diagrams.

type of kind K'. For example, the elementary data type *Integer* is of the kind $\star$. The collection type *Set* is of the kind $\star \to \star$ and the type *Set(Integer)* is of the kind $\star$. The language of types is defined as follows:

$$\tau ::= type \mid \tau(\tau_1) \mid \tau_0 \times \cdots \times \tau_n \to \tau$$

Here, a *type* is any classifier or template appearing in the contextual class diagram or the OCL standard library. The expression $\tau(\tau_1)$ expresses the type which results from instantiating a template parameter of the type τ with τ_1. The type $\tau_0 \times \cdots \times \tau_n \to \tau$ is used to express the type of properties. The type τ_0 is the type of the classifier which defines the property, the types $\tau_1, \ldots, \tau_n$ are the types of the parameters of the property. We identify attributes with operations that do not define arguments. The kinding of a type states whether a type is an elementary type or a template and is formally defined by the system shown in Table 3.1.

We write the typing rules and proofs in (the usual) natural deduction style: a rule consists of an antecedent and a consequence, which are separated by a line. The antecedent contains the properties that need to be proved in order to apply the rule and conclude its consequence. Each rule has a name, which is stated right of the line in small capitals.

$$\frac{\tau \text{ is a type or property type}}{\tau : \star}\ \text{K-Elem}$$

$$\frac{\tau \text{ is a parameterised class}}{\tau : \star \to \star}\ \text{K-Cons}$$

$$\frac{\tau_1 : K \qquad \tau : K \to K'}{\tau(\tau_1) : K'}\ \text{K-Inst}$$

Table 3.1: Kinding system

An important property of OCL is, that all types have to be defined in the contextual class diagram, with the only exception of tuple types. Tuple types (or structures) are data types. This simplifies the type checking rules a lot, because new types do not arise from the constraints to be checked. OCL expressions are checked in a context, which contains the information on variable bindings, operation declarations, and the subtype relation encoded in a class diagram. A context Γ maps variable names v to their type, or to undefined if that variable is not declared in this context. We write $\Gamma, v : \tau$ to denote the context extended by binding the variable v to τ, provided that v does not occur in Γ. We write $\Gamma(v) = \tau$ to state that v has type τ in context Γ. The context also contains the information on *type conformance*, that is, clauses of the form $\tau \leq \zeta$ derived from the

generalisation hierarchy, where τ and ζ are types and $\leq$ denotes that τ is a subtype of ζ. We write $\Gamma, \tau \leq \zeta$ to extend a context with a statement that τ is a subtype of ζ. Any context contains $\tau \leq \tau$ for every type τ occurring in the model, since the conformance relation is reflexive. If the context Γ contains the declaration $\tau \leq \zeta$ we represent this with $\Gamma \vdash \tau \leq \zeta$. We write $\Gamma \vdash t : \tau$ to denote that t is a term of type τ in the context Γ. Contexts which only differ in a different order of their declarations are considered equal. If the context is clear, we omit it in the example derivations. Finally, we assume that the context contains all declarations mandated by the OCL standard library.

The subtype relation is transitive and function application is covariant in its arguments and contra-variant in its result type. These rules are shown in Table 3.2. In rule S-Coll the notation $\Gamma \vdash C \leq \mathit{Collection}$ means that C ranges over every type which is a subtype of *Collection*. OCL defines *Bag*, *Set*, and *Sequence* as subtypes of *Collection*.

The type operator *Collection* refers to a parameterised class. The rule S-Coll-2 is not sound for classes which define operations which change the contents of the collection. The absence of side-effects in OCL expressions are a fundamental property for the validity of the rule S-Arrow and, therefore, also for S-Coll-2. The following counter-example illustrates the importance of the absence of side-effects for the type system. Consider the following fragment of C++ code:

```
class C { public: void m(double *&a) { a[0] = 1.5; } }
void main(void) { int *v = new int[1]; C *c = new C(); c->m(v); }
```

Using the rule S-Coll-2, then the call `c->m(v)` is valid, because `int` is a subtype of `double`. But within the body of `m` the assignment `a[0] = 1.5` would store a double value into an array of integers, which is not allowed. However, since we assume that each expression is free of side-effects, rule S-Coll-2 is adequate.

In OCL operations defined on collections do not alter the contents, but we cannot assume this in general; therefore, we defined these particular assumptions. The typing rules for terms are presented in Tables 3.3 and 3.4, except for the typing rule for *flatten*. The type of flatten is actually a dependent type, because it depends on the type of its argument. We present the rule in Section 3.3.5.

Rules T-True, T-False, and T-Lit assign to each literal their type. Especially, T-Lit is an axiom scheme assigning, for example, the literal 2 the type *Integer* and the literal 2.5 the type *Real*. Rule T-Coll defines the type of a collection literal. The type of a collection is determined by the declared name C and the common supertype of all its members. Rule T-Call states that if the arguments match the types of a method or a function, then the expression is well-typed and the result has the declared type. The antecedent $\Gamma \vdash C \not\leq \mathit{Collection}$ denotes that in context Γ type C is not a subtype of *Collection*.

Note that it is decidable whether a type is a subtype of another, provided the class diagram is finite. We have two cases to consider: If the two types are basic types, that is, not parameterised types. Then we can easily test whether a type is a subtype of another type. Otherwise we have to compare two *instantiations* of parameterised types.

$$\frac{\Gamma \vdash C \leq \mathit{Collection}}{\Gamma \vdash C(\tau) \leq \mathit{Collection}(\tau)}\ \text{S-Coll}$$

$$\frac{\Gamma \vdash C \leq \mathit{Collection} \quad \Gamma \vdash \tau \leq \tau'}{\Gamma \vdash C(\tau) \leq C(\tau')}\ \text{S-Coll-2}$$

$$\frac{\Gamma \vdash e : \zeta \quad \Gamma \vdash \zeta \leq \tau}{\Gamma \vdash e : \tau}\ \text{S-Sub}$$

$$\frac{\Gamma \vdash \zeta \leq \tau \quad \Gamma \vdash \tau \leq \upsilon}{\Gamma \vdash \zeta \leq \upsilon}\ \text{S-Trans}$$

$$\frac{\Gamma \vdash \tau_0 \leq \zeta_0, \Gamma \vdash \zeta_1 \leq \tau_1, \cdots, \Gamma \vdash \zeta_n \leq \tau_n \quad \Gamma \vdash \tau \leq \zeta}{\Gamma \vdash \tau_0 \times \tau_1 \times \cdots \times \tau_n \to \tau \leq \zeta_0 \times \zeta_1 \times \cdots \times \zeta_n \to \zeta}\ \text{S-Arrow}$$

Table 3.2: Definition of type conformance

Since these two are finite terms, we can recursively check whether the type arguments are subtypes of each other, and, if this succeeds, whether the type operators are related in the class diagram.

A similar rule for collection calls is given by T-CCall. Recall that the antecedent $\Gamma \vdash C \leq \mathit{Collection}$ states that the type of t_0 has to be a collection type. Rule T-Cond defines the typing of a condition. If the condition e_0 has the type *Boolean* and the argument expressions e_1 and e_2 have a common supertype τ, then the conditional expression has that type τ. Rule T-Iterate gives the typing rule for an iterate expression. First, the expression we are iterating over has to be a collection. Then the accumulator has to be initialised with an expression of the same type. Finally, the expression we are iterating over has to be an expression of the accumulator variables type in the context which is extended by the iterator variable and the accumulator variable. Rule T-Let defines the rule for a let expression of the OCL 2.0 standard. Rule T-RLet allows a let-expression where the user can define functions and use mutual recursion. There we add all variables declared by the let expression to context Γ in order to obtain context Γ'. Each expression defined has to be well typed in the context extended by the formal parameters of the definition. Finally, the expression in which we use the definitions has to be well-typed in the context Γ'.

This type system is a faithful representation of the type system given in [113], but we have omitted the typing rules for the boolean connectives, as they are given by Cengarle and Knapp in [23], because each expression using a boolean connective can be rewritten to an operation call expression, for example, *a and b* is equivalent to

$$\frac{}{\Gamma \vdash \text{true} : \mathit{Boolean}}\ \text{T-True}$$

$$\frac{}{\Gamma \vdash \text{false} : \mathit{Boolean}}\ \text{T-False}$$

$$\frac{}{\Gamma \vdash l : \tau_l}\ \text{T-Lit}$$

$$\frac{\Gamma(v) = \tau}{\Gamma \vdash v : \tau}\ \text{T-Var}$$

$$\frac{C \in \{\mathit{Bag}, \mathit{Set}, \mathit{Sequence}\} \quad \Gamma \vdash e_1 : \tau \quad \cdots \quad \Gamma \vdash e_n : \tau}{C\{e_1, \ldots, e_n\} : C(\tau)}\ \text{T-Coll}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \cdots \quad \Gamma \vdash e_n : \tau_n}{\mathit{Tuple}\{a_1 = e_1, \ldots, a_n = e_n\} : \mathit{Tuple}\{a_1 : \tau_1, \ldots, a_n : \tau_n\}}\ \text{T-Tuple}$$

$$\frac{\Gamma \vdash e_0 : \mathit{Boolean} \quad \Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau}{\textbf{if } e_0 \textbf{ then } e_1 \textbf{ else } e_2 \textbf{ endif} : \tau}\ \text{T-Cond}$$

$$\frac{\begin{array}{c}\Gamma \vdash t_0 : \zeta \quad \Gamma \vdash t_1 : \zeta_1, \cdots, \Gamma \vdash t_n : \zeta_n \\ \Gamma \vdash m : \zeta \times \zeta_1 \times \cdots \times \zeta_n \to \tau \quad \Gamma \vdash \zeta \not\leq \mathit{Collection}\end{array}}{\Gamma \vdash t_0.m(t_1, \ldots, t_n) : \tau}\ \text{T-Call}$$

$$\frac{\begin{array}{c}\Gamma \vdash t_0 : C(\zeta) \quad \Gamma \vdash t_1 : \zeta_1, \cdots, \Gamma \vdash t_n : \zeta_n \\ \Gamma \vdash m : C(\zeta) \times \zeta_1 \times \cdots \times \zeta_n \to \tau \quad \Gamma \vdash C \leq \mathit{Collection}\end{array}}{\Gamma \vdash t_0 \to m(t_1, \ldots, t_n) : \tau}\ \text{T-CCall}$$

$$\frac{\Gamma \vdash t_0 : C(\tau) \quad \Gamma \vdash t : \zeta \quad \Gamma, v : \tau, a : \zeta \vdash e : \zeta \quad \Gamma \vdash C \leq \mathit{Collection}}{\Gamma \vdash t_0 \to \mathit{iterate}(v; a = t \mid e) : \zeta}\ \text{T-Iterate}$$

Table 3.3: Typing rules for OCL

$$\frac{\Gamma \vdash t_0 : \tau_0 \quad \Gamma, v_0 : \tau_0 \vdash t : \tau}{\Gamma \vdash \textbf{let } v_0 : \tau_0 = t_0 \textbf{ in } t \ : \tau} \text{ T-Let}$$

$$\frac{\begin{array}{l}\Gamma' = \Gamma, \quad v_0 : \tau_{0,0} \times \cdots \times \tau_{0,m_0} \to \tau_0, \\ \qquad\quad \ldots, \\ \qquad\quad v_n : \tau_{n,0} \times \cdots \times \tau_{n,m_n} \to \tau_n \\ \Gamma', v_{0,0} : \tau_{0,0}, \ldots, v_{0,m_0} : \tau_{0,m_0} \vdash t_0 : \tau_0 \\ \qquad\qquad\qquad \vdots \\ \Gamma', v_{n,0} : \tau_{n,0}, \ldots, v_{n,m_n} : \tau_{n,m_n} \vdash t_n : \tau_n \\ \qquad\qquad \Gamma' \vdash t : \tau\end{array}}{\begin{array}{ll}\Gamma \vdash & \textbf{let} \quad v_0(v_{0,0}, \ldots, v_{0,m_0}) : \tau_0 = t_0, \\ & \qquad \ldots, \\ & \qquad v_n(v_{n,0}, \ldots, v_{n,m_n}) : \tau = t_n \\ & \textbf{in } t \ : \tau\end{array}} \text{ T-RLet}$$

Table 3.4: Typing rules for OCL (continued)

a.and(*b*). We do not have a rule for the undefined value, as presented in [23], because the OCL 2.0 proposal does not define a literal for undefined [113, pp. 48–50].[4]

Within the UML 2.0 standard [111] and the OCL 2.0 standard [113] methods are redefined co-variantly. We assume that some kind of multi-method semantics for calls of these methods is intended. These redefinitions are not explicitly treated in the OCL 2.0 type system; they can, however, be treated as overloading a method, and, hence, be modelled with union-types in our system (see Section 3.3.2), as suggested in, for example, by Pierce [126, p. 340].

Also note that our type system makes use of the largest common supertype only implicitly, whereas it is explicitly used in other papers. It is hidden in the type conformance rules of Table 3.2. An example of where we use the largest common supertype can be found in Section 3.3.1. The rules presented here are not designed for a type-checking algorithms, but for deriving well-typedness. Therefore, the type system presented here lacks the unique typing property, but it is adequate and decidable.[5]

[4]Note that *OclInvalid* is the type of any undefined expression, which does not have a literal for its instances [113, p. 133]. Calling the property *oclIsUndefined*(), defined for any object, is preferred, because any other property call results in an instance of *OclInvalid*.

[5]For a type system with unique typing, first rules computing a normal form for each type have to be defined. These rules have to form a confluent rewriting system, otherwise the type system is not decidable. Then an equality rule for types is needed. Finally, a priority on the rules has to be declared and it has to be asserted that rules of the same priority have mutually exclusive antecedents. Details on this procedure can be found in [30] or in [145].

Proposition 3.1. *The type system is* adequate*, that is, for any OCL expression e we have: if* $e : \tau$ *can be derived in the type system, then e is evaluated to a result of a type conforming to* τ*.*

The type system is decidable*, that is, there exists an algorithm which either derives a type* τ *for any OCL expression e or reports that no type can be derived for e.*

A proof of this proposition has been provided by Cengarle and Knapp in [23].

We use the definitions of this section for the discussion of its limitations in the following sections.

3.3 Extensions

In this section, we propose various extensions to the type system of OCL which help to use OCL earlier in the development of a system and to write more expressive constraints. We introduce intersection types, union types, and bounded operator abstraction to the type system of OCL. Intersection types, which express that an object is an instance of all components of the intersection type, are more robust with respect to transformations of the contextual class diagram. Union types, which express that an object is an instance of at least one component of the union type, admit more constraints that have a meaning in OCL to be well-typed. Parametric polymorphism extends OCL to admit constraints on templates without requiring that the template parameter is bound. Bounded parametric polymorphism allows one to specify assumptions on a template parameter. Together, our extensions result in a more flexible type system which admits more OCL constraints to be well typed without sacrificing adequacy or decidability.

3.3.1 Intersection Types

Consider the following constraint of class *Obs* in Figure 3.1:

$$\textbf{context}\ \mathit{Obs}\ \textbf{inv} : a \to \mathit{union}(b).m() \to \mathit{forAll}(x \mid x > 1)$$

This is a simple constraint which asserts that the value returned by *m* for each element in the collection of a and b is always greater than 1. We show that it is well-typed in OCL using the type system of Section 3.2.

$$\dfrac{\dfrac{\dfrac{a : \mathit{Bag}(D) \quad b : \mathit{Bag}(C)}{a \to \mathit{union}(b) : \mathit{Bag}(A)}\ \text{T-CC}_{\text{ALL}}}{a \to \mathit{union}(b).m() : \mathit{Bag}(\mathit{Integer})}\ \text{T-C}_{\text{ALL}}}{a \to \mathit{union}(b).m() \to \mathit{forAll}(x \mid x > 1) : \mathit{Boolean}}\ \text{T-CC}_{\text{ALL}}$$

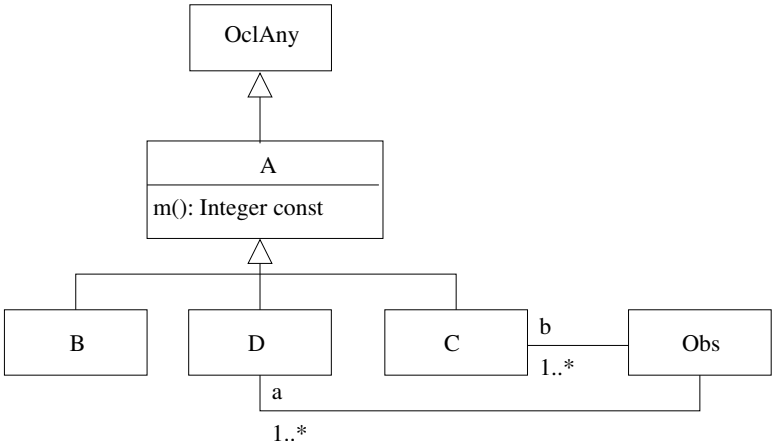


Figure 3.1: A simple initial class diagram

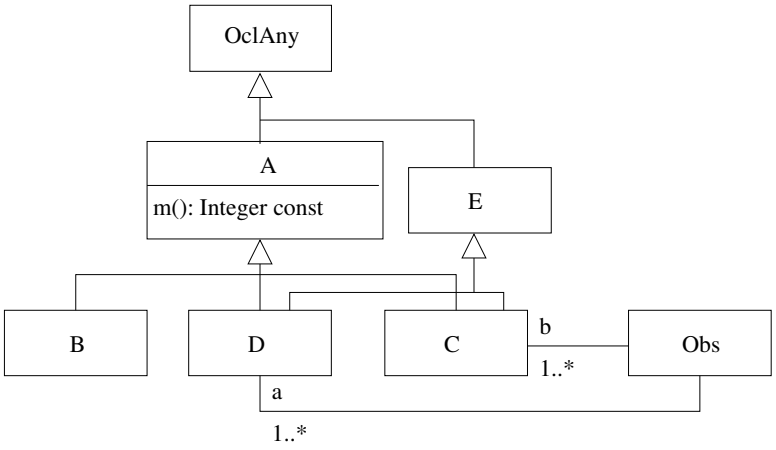


Figure 3.2: The same diagram after a change

Now consider the following question: What happens to the constraint if we change the class diagram to the one in Figure 3.2, which introduces a new class E that implements the common functions of classes C and D? The meaning of the constraint is not affected by this change. However, the OCL constraint is not well-typed anymore, as this derivation shows, where the type annotation $\lightning$ is used to state that the type system is not able to derive a type for the expression.[6]

$$\dfrac{\dfrac{a : Bag(D) \quad b : Bag(C)}{a \to union(b) : Bag(A)}\,\text{T-CC}_{\text{ALL}}}{a \to union(b).m() : \lightning}$$

The problem is, that the OCL type system chooses the unique and most precise supertype of a and b to type the elements of $a \to union(b)$, which now is *OclAny*, because we now have to choose one of A, E, and *OclAny*, which are the supertypes of C and D. Neither A nor E are feasible, because the type of the expression has to be chosen now. Hence, we are forced to choose *OclAny*. To avoid this problem constraints should be written once the contextual class diagram does not change anymore. Otherwise, all constraints have to be updated, which if it is done by hand is a time consuming and error-prone task.

The mentioned insufficiency of the type system can be solved in two ways: We can implement a transformation which updates all constraints automatically after such a change, or we introduce a more permissive type system for OCL. Because an automatic update of all constraints entails an analysis of all constraints in *the same way* as performed by the more permissive type system, therefore we extended the type system and leave the constraints unchanged.

The proposed extension is the introduction of intersection types. An intersection type, written $\tau \wedge \tau'$ for types τ and τ' states that an object is of type τ and τ'. Because $\wedge$ is both an associative and commutative operator, we introduce the generalised intersection $\bigwedge_{\tau \in \mathcal{T}} \tau$. In this chapter $\mathcal{T}$ is always a *finite* set of types.

Recall, that the generalisation hierarchy is a partially ordered set. Intersection types have been defined, such that they are compatible with generalisation. It can be easily shown, that the set of types and their intersection types form a lattice with the empty intersection type $\bigwedge \emptyset$ as the top type (cf. Pierce [126]).

The empty intersection $\bigwedge \emptyset$ is a subtype of any other type and does not have any instances; therefore, it is equivalent to *OclVoid*, which also does not have any instances and is a subtype of any other type.

Intersection types are very useful to explain multiple inheritance, a relation that has been investigated by Cardelli and Wegener [21], Pierce [125], and Compagnoni [31].

We add the rules of Table 3.5 to the type system, which introduces intersection types into the type hierarchy. The rule S-InterLB and S-Inter formalise the notion that a type

[6] Recall, that we do not explicitly write the context in examples if it is clear.

τ belongs to both types, and that $\wedge$ corresponds to the order-theoretic meet. The rule S-InterA allows for a convenient interaction with operation calls and functions. This

$$\frac{\tau \in \mathcal{T}}{\Gamma \vdash \bigwedge_{\tau' \in \mathcal{T}} \tau' \leq \tau} \text{ S-InterLB}$$

$$\frac{\Gamma \vdash \tau \leq \tau' \text{ for all } \tau' \in \mathcal{T}}{\Gamma \vdash \tau \leq \bigwedge_{\tau' \in \mathcal{T}} \tau'} \text{ S-Inter}$$

$$\frac{}{\Gamma \vdash \left(\bigwedge_{\tau' \in \mathcal{T}} (\tau \to \tau') \right) \leq \left(\tau \to \bigwedge_{\tau' \in \mathcal{T}} \tau' \right)} \text{ S-InterA}$$

Table 3.5: Intersection types

extension of the type system already solves the problem raised for the OCL constraint in the context of Figure 3.2, as this derivation demonstrates:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{a : Bag(D) \quad D \leq A \quad D \leq E}{a : Bag(A \wedge E)} \text{ S-Inter} \qquad \dfrac{b : Bag(C) \quad C \leq A \quad C \leq E}{b : Bag(A \wedge E)} \text{ S-Inter}}{a \to union(b) : Bag(A \wedge E)} \text{ T-CCall}}{a \to union(b) : Bag(A)} \text{ S-InterLB}}{a \to union(b).m() : Bag(Integer)} \text{ T-Call}}{a \to union(b).m() \to forAll(x \mid x > 1) : Boolean} \text{ T-CCall}$$

The extension of the OCL type system with intersection types is sufficient to deal with transformations which change the class hierarchy by moving common code of a class into a new super-class. This extension is also safe, and does not change the decidability of the type system.

3.3.2 Union Types

Union types are dual to intersection types, and can be used to address type-checking of overloaded operators. Union types also solve type checking problems for collection literals and the union operation of collections in OCL. We explain this by the class diagram in Figure 3.3. Consider the expression **context** C **inv** : $Set\{self.a, self.b\}.m() \to$

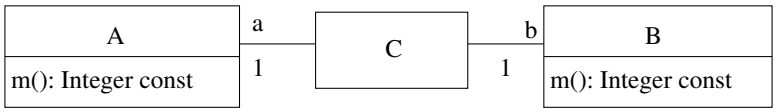


Figure 3.3: A simple example class diagram

$size() > 1$ on this class diagram.[7] Assuming that the multiplicities of the associations are 1, we have:

$$\dfrac{\dfrac{a : A \quad b : B}{Set\{a, b\} : Set(OclAny)}\,\text{T-Coll}}{Set\{a, b\}.m() : \lightning}$$

even though both types A and B define the property $m(x : Integer) : Integer$. Here, it is desirable to admit the constraint as well-typed, because it also has a meaning in OCL. Using intersection types does not help here, because stating that a and b have the type $A \wedge B$ is not adequate.

Instead, we want to judge that a and b have the type A *or* B. For this purpose we propose to introduce the union type $A \vee B$. A union type states, that an object is of type A or it is of type B. Again, $\vee$ is associative and commutative, so we introduce the generalised union $\bigvee_{\tau \in \mathcal{T}} T$. The type $\bigvee \emptyset$ is the universal type, a supertype of *OclAny*, of which any object is an instance. Union types are characterised by the rules in Table 3.6. Rules S-UnionUB and S-Union formalise the fact that a union type is the least upper

$$\frac{\tau \in \mathcal{T}}{\tau \leq \bigvee_{\tau' \in \mathcal{T}} \tau'}\,\text{S-UnionUB}$$

$$\frac{\tau' \leq \tau \text{ for all } \tau' \in \mathcal{T}}{\bigvee_{\tau' \in \mathcal{T}} \tau' \leq \tau}\,\text{S-Union}$$

$$\frac{}{\left(\bigwedge_{\tau' \in \mathcal{T}} (\tau' \to \tau)\right) \leq \left((\bigvee_{\tau' \in \mathcal{T}} \tau') \to \tau\right)}\,\text{S-UnionA}$$

Table 3.6: Rules for union types

bound of two types. Note that it only makes sense to use a property on objects of $A \vee B$

[7] Note that $a : A$ and $b : B$, and both classes define a method $m()$ returning an *Integer*. However, in this case the intended meaning of the constraint is $Set\{self.a.m(), self.b.m()\} \to size() > 1$, which is well-defined.

that are defined for A and B. This is stated by the rule S-UnionA.

Using our extended type system, we can indeed derive that our example has the expected type.

$$\dfrac{\dfrac{\dfrac{\dfrac{a : A \quad b : B}{Set\{a, b\} : Set(A \vee B)}\,\text{S-UnionUB} \qquad \dfrac{m : A \to Integer \quad m : B \to Integer}{m : A \vee B \to Integer}\,\text{S-UnionA}}{Set\{a, b\}.m() : Bag(Integer)}\,\text{T-Call}}{Set\{a, b\}.m() \to size() : Integer}\,\text{T-CCall}}{Set\{a, b\}.m() \to size() > 0 : Boolean}\,\text{T-Call}$$

3.3.3 Parametric Polymorphism

UML 2.0 provides the user with templates (see [111, Section 17.5, pp. 541ff.]), which are also called generics or parameterised classes, which are functions from types or values to types, that is, they take a type as an argument and return a new type. We first consider the case where the parameter of a class ranges over types. Adequate support for parametric polymorphism in the specification language is again highly useful, as the proof of a property of a template carries over to all its instantiations, which is explained in Wadler's beautiful paper "Theorems for Free!" [152]. The OCL standard library contains the collection types, which are indeed examples of generic classes.

The type of a generic class, which is indeed a type operator, that is a function which operates on types, is written as $\Lambda\tau \leq \zeta : \upsilon$. A type operator of this form creates a type $\upsilon[\tau/\tau']$ if it is applied to a type τ', which has to be a subtype of ζ. Therefore, one speaks of bounded operator abstraction (see also Section 3.3.4). Because parameterised classes define operations and methods, the methods are terms which are parameterised in the same way as their class, that is, the body b of a method is parameterised by τ. Therefore, we speak of parametric polymorphism.

Example 3.1. Consider the OCL type operator $\Lambda\tau \leq OclAny : Set(\tau)$ (we have made the type abstraction explicit in the type name). Independent of the concrete type instantiated for τ, we can easily reason about the behaviour of each operation defined for $\Lambda\tau \leq OclAny : Set(\tau)$, because, for example, the definition of *count* given in Section 2.3.4 does not depend on any property of the objects contained in an instance of an instance of $\Lambda\tau \leq OclAny : Set(\tau)$. ♦

We have not found yet how the parameter of a template, which is defined in the class diagram, is integrated into OCL's type system. In fact, it is not defined in the proposal how the environment has to be initialised in order to parse expressions according to the rules of Chapter 4 of [113]. For example, consider the following constraint:

$$\begin{aligned}&\textbf{context } Sequence :: excluding(o : \tau) : Sequence(\tau)\\ &\textbf{post} : result = self \to iterate(e; a : Sequence(\tau) = Sequence\{\} \mid \\ &\quad \textbf{if } e = o \textbf{ then } a \textbf{ else } a \to append(o) \textbf{ endif})\end{aligned} \tag{3.1}$$

To what does τ refer to? Currently, τ is not part of the type environment, because it is neither a classifier nor a state but an instance of *TemplateParameter* in the UML meta-model. This constraint is, therefore, not well-typed. But it is worthwhile to admit constraints like (3.1), because this constraint is valid for any instantiation of the parameter τ.

UML 2.0 allows different kinds of template parameters: parameters ranging over classifiers, parameters ranging over value specifications, and parameters ranging over features (properties and operations). In this chapter, we only consider parameters ranging over classifiers.

We propose to extend the environment such that Γ contains the kinding judgement $\tau \in \star$ if τ is the parameter of a template. This states that the parameter of a template is a type. Also note that the name of the template classifier alone is not of the kind $\star$ but of some kind $\star \to \cdots \to \star$, depending on the number of type parameters. Additionally, we give the following type checking rules for templates in Table 3.7. These rules generalises the conforms-to relation previously defined for collection types only.

$$\frac{\Gamma \vdash \tau : K \to K' \quad \Gamma \vdash \tau' : K \to K' \quad \Gamma \vdash \tau'' : K \quad \Gamma \vdash \tau \leq \tau'}{\Gamma \vdash \tau(\tau'') \leq \tau'(\tau'')}\ \text{S-InstSub}$$

$$\frac{\Gamma \vdash \tau : K \to K' \quad \Gamma \vdash \tau' : K \quad \Gamma \vdash \tau'' : K \quad \Gamma \vdash \tau' \leq \tau''}{\Gamma \vdash \tau(\tau') \leq \tau(\tau'')}\ \text{S-InstSub-2}$$

Table 3.7: Subtyping rules for parametric polymorphism

The rule S-InstSub states that if a template class τ is a subtype of another template class τ', then τ remains a subtype of τ' for any class τ'' bound to the parameter. This rule is always adequate. The rule T-InstSub-2 states that for any template class τ and any types τ' and τ'' such that τ' is a subtype of τ'', then binding τ' and τ'' to the type parameter in τ preserves this relation. Both rules generalise S-Coll and S-Coll-2 to all parameterised classes.

Because rule S-InstSub-2 generalises rule S-Coll-2, it is not always safe, for the same reasons that have been described in Section 3.2.

3.3.4 Bounded Operator Abstraction

While parametric polymorphism in the form of templates is useful in itself, certain properties still cannot be expressed directly as types but have to be expressed in natural language. For example, in the OCL standard the collection property `sum()` of set has the following specification:

The addition of all elements in *self*.[8] Elements must be of a type supporting the + operation. The + operation must take one parameter of type τ and be both associative: $(a + b) + c = a + (b + c)$, and commutative: $a + b = b + a$. Integer and Real fulfil this condition.

Formally, the post condition of sum does not type check, because a type checker has no means to deduce that T indeed implements the property + as specified. The information can be provided in terms of bounded polymorphism, where the type variable is bounded by a super type. The properties of + can be specified in an abstract class (or interface), say *Sum*, and the following constraints:

$$\begin{aligned}&\textbf{context } \mathit{Sum}\\&\textbf{inv} : \mathit{self}.\mathit{typeOf}().\mathit{allInstances}()\rightarrow\mathit{forAll}(a, b, c \mid\\&\qquad a + (b + c) = (a + b) + c)\\&\textbf{inv} : \mathit{self}.\mathit{typeOf}().\mathit{allInstances}()\rightarrow\mathit{forAll}(a, b \mid a + b = b + a)\end{aligned} \tag{3.2}$$

In Equation (3.2) the property *typeOf*() is supposed to return the run-time type of the object represented by *self*. It is important to observe that we cannot write *Sum.allInstances*, because the type implementing *Sum* need not provide an implementation of + which work uniformly on *all* types implementing *Sum*. For example, we can define + on *Real* and on *Vectors* of *Real*s, but it may not make sense to implement an addition operation of vectors to real which returns a real. So we do not want to force the modeller to do this. The purpose of *Sum* is to specify that a classifier provides an addition which is both associative and commutative.

When *Sum* is a base class of a classifier τ, and we have a collection of instances of τ, then we also know that the property *sum* is defined for this classifier. So *Sum* is a lower bound of the types of τ. Indeed, the signature of *Collection* :: *sum* can be specified by $\mathit{Collection}(T \leq \mathit{Sum}) :: \mathit{sum}() : T$, which expresses the requirements on τ.

Syntactically, we express a bounded template using the notation $C(\tau \leq \zeta)$, defining a type of kind $\Pi\tau \leq \zeta \rightarrow \star$, provided that ζ is of kind $\star$. Here the new kind $\Pi\tau \leq \zeta$ states that τ has to be a subtype of ζ to construct a new type, otherwise, the type is not well-kinded.

Bounded operator abstraction is highly useful in designing object-oriented programs. They enable to express assumptions about the interfaces of types, from which the implementation of the class abstracts using parametric polymorphism, in defining a bound. Many examples of how to design systems using bounded operator abstraction have been described in Bertrand Meyers "Object-Oriented Software Construction" [98]. This form of bounded operator abstraction has been implemented, for example, in the Eiffel programming language [97].

[8] Recall that OCL views the first operand of an operation as the receiver of a message. Here *self* represents the collection, whose elements are to be summed up.

Observe that abstracted operators are not comparable using the subtype relation, that is, we have neither

$$\Lambda T \preceq OclAny : Collection(T) \preceq \Lambda T \preceq OclAny : Set(T)$$

nor

$$\Lambda T \preceq OclAny : Set(T) \preceq \Lambda T \preceq OclAny : Collection(T)$$

Therefore, no additional rules have to be introduced.

3.3.5 Flattening and Accessing the Run-Time Type of Objects

Quite often it is necessary to obtain the type of an object and compare it. OCL provides some functions which allow the inspection and manipulation of the run-time type of objects. To test the type of an object it provides the operations *oclIsTypeOf()* and *oclIsKindOf()*, and to *cast* or coerce an object to another type it provides *oclAsType()*. In OCL, we also have the type *OclType*, of which the values are the names of all classifiers appearing in the contextual class diagrams.[9] The provided mechanisms are not sufficient, as the specification of the *flatten()* operation shows (see [113]):

$$\begin{array}{l}\textbf{context } Set :: flatten() : Set(\tau_2)\\ \textbf{post} : result = \textbf{if } self.type.elementType.oclIsKindOf(\text{CollectionType})\\ \quad \textbf{then } self \rightarrow iterate(c; a : Set() = Set\{\} \mid a \rightarrow union(c \rightarrow asSet()))\\ \quad \textbf{else } self\\ \quad \textbf{endif}\end{array} \tag{3.3}$$

This constraint contains many errors. First, the type variable τ_2 is not bound in the model (see Section 3.3.3 for the meaning of binding), so it is ambiguous whether τ_2 is a classifier appearing in the model or a type variable. Next, self is an instance of a collection kind, so the meaning of *self.type* is actually a shorthand for $self \rightarrow collect(type)$, and there is no guarantee that each instance of the collection defines the property *type*. Of course, the intended meaning of this sub-expression is to obtain the element-type of the members of *self*, but one cannot access the environment of a variable from OCL. Next, the type of the accumulator in the iterate expression is not valid, *Set* requires an argument, denoting the type of the elements of the accumulator set (one could use τ_2 as the argument).

The obvious solution, to allow the type of an expression depending on the type of other expressions, poses a serious danger: If the language or the type system is too permissive in what is allowed as a type, we cannot algorithmically decide, whether

[9] The type *OclType* will be removed but still occurs in the proposal.

a constraint is well-typed or not. But decidability is a desirable property of a type-system. Instead, we propose to treat the *flatten()* operation as a kind of literal, like *iterate* is treated. For *flatten*, we introduce the following two rules:

$$\frac{e : C(\tau) \qquad C \leq \mathit{Collection} \qquad \tau \leq \mathit{Collection}(\tau')}{e \to \mathit{flatten}() : C(\tau')}\ \text{T-FLAT}$$

$$\frac{e : C(\tau) \qquad C \leq \mathit{Collection} \qquad \tau \not\leq \mathit{Collection}(\tau')}{e \to \mathit{flatten}() : C(\tau)}\ \text{T-NFLAT}$$

The rule T-FLAT covers the case where we may flatten a collection, because its element type conforms to a collection type with element type τ'. In this case, τ' is the new collection type. The rule T-NFLAT covers the case where the collection e does not contain any other collections. In this case, the result type of *flatten* is the type of collection e.

These rules encode the following idea: For each collection type we define an *overloaded* version of flatten. As written in Section 3.3.2, we are able to define the type of any overloaded operation using a union type. However, using this scheme directly yields infinitary union types, because the number of types for which we have to define a flatten operation is not bounded. The price for this extension is decidability [10].

The drawback of this extension is that the meaning of the collection cannot be expressed in OCL, because we have no way to define τ' in OCL. The advantage is, that the decidability of the type system extended in this way is not affected.

3.4 Adequacy and Decidability

In this section, we summarise the most important results concerning the extended type system. This means that if the type system concludes that an OCL-expression has type τ, then the result of evaluating the expression yields a value of a type that conforms to τ. The type system is adequate and decidable. For the (operational) semantics of OCL we use the one defined in Chapter 2.

Theorem 3.2 (Adequacy)**.** *Let Γ be a context, e an OCL expression, and τ a type of kind $\star$ such that $\Gamma \vdash e : \tau$. Then the value of e is either undefined or it conforms to τ.*

Proof. The proof is by structural induction on the construction of e We only treat some example cases.

1. Assume $e = \ell$ for some literal and $e : \mathit{Real}$. By the literal axiom of Definition 2.19 it follows that $\mathrm{eval}(D, \sigma, \overleftarrow{\sigma}, \Gamma)(\ell) = \ell \in \mathbb{R}$ for all $D, \sigma, \overleftarrow{\sigma}, \Gamma$.

2. Assume $e = \textbf{if } e_0 \textbf{ then } e_1 \textbf{ else } e_2 \textbf{ endif} : \tau$ and the induction hypothesis is true for e_0, e_1, and e_3. By rule T-COND and the induction hypothesis we have $e_0 : \mathit{Boolean}$ and $\mathrm{eval}(D, \sigma, \overleftarrow{\sigma}, \Gamma)(e_0) \in \mathbb{B}_{\mathcal{J}_{\mathbb{B}}}$.

a) If $\text{eval}(D, \sigma, \overleftarrow{\sigma}, \Gamma)(e_0) = \text{true}$, and, by induction hypothesis $\Gamma \vdash e_1 : T$, then $\text{eval}(D, \sigma, \overleftarrow{\sigma}, \Gamma)(e) = \text{eval}(D, \sigma, \overleftarrow{\sigma}, \Gamma)(e_1)$, which is a value that conforms to T.

b) If $\text{eval}(D, \sigma, \overleftarrow{\sigma}, \Gamma)(e_0) = \text{false}$, and, by induction hypothesis $\Gamma \vdash e_2 : T$, then $\text{eval}(D, \sigma, \overleftarrow{\sigma}, \Gamma)(e) = \text{eval}(D, \sigma, \overleftarrow{\sigma}, \Gamma)(e_2)$, which is a value that conforms to T.

c) If $\text{eval}(D, \sigma, \overleftarrow{\sigma}, \Gamma)(e_0) = \mathcal{J}_{\mathbb{B}}$, then $\text{eval}(D, \sigma, \overleftarrow{\sigma}, \Gamma)(e) = \mathcal{J}_T$.

3. Assume $e = e_0 \to \mathit{iterate}(v; a = e_1 \mid e_2)$, and as an induction hypothesis, that we have $\Gamma \vdash e_0 : \mathit{Collection}(T)$, $v : T$, $a : T'$, $\Gamma \vdash e_1 : T'$ and $\Gamma, v : T, a : T' \vdash e_2 : T'$. Also, as an induction hypothesis, assume, that $\text{eval}(D, \sigma, \overleftarrow{\sigma}, \Gamma)(e_0) \in \mathfrak{D}(\mathit{Collection}(T))$ and $\text{eval}(D, \sigma, \overleftarrow{\sigma}, \Gamma)(e_1) \in \mathfrak{D}(T')$, and it holds that

$$\lambda x.\lambda y.\, \text{eval}(D, \sigma, \overleftarrow{\sigma}, \Gamma\{v \mapsto x, a \mapsto y\})(e_2) \in \mathfrak{D}(T \to T' \to T') \quad .$$

Then we distinguish three cases:

a) If $\text{eval}(D, \sigma, \overleftarrow{\sigma}, \Gamma)(e_0)$ is not finite, then $\text{eval}(D, \sigma, \overleftarrow{\sigma}, \Gamma)(e_0) = \mathcal{J}'_T$.

b) If $\text{eval}(D, \sigma, \overleftarrow{\sigma}, \Gamma)(e_0)$ is empty, then

$$\text{eval}(D, \sigma, \overleftarrow{\sigma}, \Gamma)(e) = \text{eval}(D, \sigma, \overleftarrow{\sigma}, \Gamma)(e_1) \quad .$$

c) Else, the induction hypothesis $\lambda x.\lambda y.\, \text{eval}(D, \sigma, \overleftarrow{\sigma}, \Gamma\{v \mapsto x, a \mapsto y\})(e_2) \in \mathfrak{D}(T \to T' \to T')$ implies

$$(\lambda x.\lambda y.\, \text{eval}(D, \sigma, \overleftarrow{\sigma}, \Gamma\{v \mapsto x, a \mapsto y\})(e_2))(x)(y) \in \mathfrak{D}(T')$$

for all $x \in \text{eval}(D, \sigma, \overleftarrow{\sigma}, \Gamma)(e_0)$ and $y = \text{eval}(D, \sigma, \overleftarrow{\sigma}, \Gamma)(e_1)$.

The proofs for the other cases have a very similar structure. □

The proof presented here is very similar to the one presented Cengarle and Knapp in [23], but Cengarle and Knapp use a slightly different type system and a quite different semantics.

Theorem 3.3 (Decidability of Type Checking)**.** *For any context* Γ *and any OCL expression* e *and type* τ *of kind* $\star$*, it is decidable whether* $\Gamma \vdash e : \tau$.

The proof of this theorem is a consequence of the theorems by Compagnoni [30] and Steffen [145]. The key components of the proof are: Infer a minimal type T' for e in Γ using a decidable procedure, and check whether $\Gamma \vdash T' \leq T$, which has to be decidable.

We have used the type inference algorithm of Definition 4.81 in [30], which has been proved decidable in this paper. For the case of checking subtyping, we point out

that OCL and UML only allow types of kind $\star$ to be compared. This reduces our type system to a special case of [30].

Constraints for methods defined in parameterised classes are handled by instantiating an arbitrary type T for the type variable τ and by adding the axiom T is a subtype of the lower bound defined in the abstraction.

We do not give the proof of Theorem 3.3 here, because our proof does not provide any new insight over the proof given by Compagnoni, and her proof is about 50 pages.

Our algorithm is an adaption of [30] and [145] to handle union types. It is simpler, because we only have a form of bounded operator abstraction, where type abstractions are not comparable, which simplifies the subtyping problem, and polymorphism is ML-like.

Another result is concerning incompleteness of the type checking procedure. By this we mean that if e is an OCL expression the type system will not compute the most precise type of e, but one of its supertypes. The reason for incompleteness is the following: If e is a constraint whose evaluation does not terminate, its most specific type is *OclVoid*. But we cannot decide whether the evaluation of a constraint will always terminate.

Indeed, the type system presented in this chapter addresses many features of UML and OCL: multiple inheritance, operator overloading, parameterised classifiers, and bounded operator abstraction. And the type-checking problem is still decidable for this type system. If we, for example, also add checking for value specifications of method specifications of parameterised classes to the type system, the type theory would correspond to the calculus of constructions, which is undecidable, as described by Coquand [35]. Such type systems indeed form the theoretical foundation of interactive theorem provers.

3.5 Related Work and Conclusions

A type system for OCL has been presented by Clark in [25], by Richter and Gogolla in [135], and by Cengarle and Knapp in [23]. In Section 3.2 we summarised these results and give a formal basis for our proposal.

A. Schürr has described an extension to the type system of OCL [141], where the type system is based on set approximations of types. These approximations are indeed another encoding of intersection and union types. His algorithm does not work with parameterised types and bounded polymorphism, because the normal forms of types required for the proof of Theorem 3.3 cannot be expressed as finite set approximations. We extended OCL's type system to also include polymorphic specifications for OCL constraints, which is not done by Schürr.

Our type system is a special case of the calculus $F^{\omega}_{\wedge}$. This system is analysed in [30], where a type checking algorithm is given. This calculus is a conservative extension of $F^{\omega}_{\leq}$. M. Steffen has described a type checking algorithm for $F^{\omega}_{\leq}$ with polarity infor-

mation [145]. Our type system does not allow type abstractions in expressions and assumes that all type variables are universally quantified in prenex form.

We have presented extensions to the type system for OCL, which admits a larger class of OCL constraints to be well-typed. Furthermore, we have introduced extensions to OCL, which allow to write polymorphic constraints.

The use of intersection types simplifies the treatment of multiple inheritance. This extension makes OCL constraints robust to changes in the underlying class diagram, for example, refactoring by moving common code into a superclass. Intersection types are therefore very useful for type-checking algorithms for OCL. Union types simplify the treatment of collection literals, model operator overloading elegantly, and provides unnamed supertypes for collections and objects. Parametric polymorphism as introduced by UML 2.0's templates is useful for modelling. We described how polymorphism may be integrated into OCL's type system and provided a formal basis in type checking algorithms. Bounded parametric polymorphism is even more useful, because it provides the linguistic means to specify assumptions on the type of the type parameters.

We have proposed typing rules for certain functions which can not be formally expressed in OCL. We have shown that this type system is sound, adequate, and decidable.

Chapter 4

Formalising UML Models and OCL Constraints in PVS

The Object Constraint Language (OCL) is not yet widely adopted in industry, because proper and integrated tool support is lacking. We describe a prototype tool, which analyses the syntax and semantics of OCL constraints together with a UML model and translates them into the language of the theorem prover PVS. This defines a formal semantics for both UML and OCL, and enables the formal verification of systems modelled in UML. We handle the problematic fact that OCL is based on a three-valued logic, whereas PVS is only based on a two-valued one.

4.1 Introduction

Today, many tools are available which support developing systems using UML's notations, ranging from syntactic analysers (Warmer [153]) to simulators (OCLE [32]), compilers enabling run-time checking of specifications (Hussmann, Demuth, and Finger [68]), model checkers (Latella, Majzik, and Massnik [92] or del Mar Gallardo, Merino, and Pimentel [47]), and integrations with theorem provers (Aredo [6]). However, until now no tool integrates verification and validation of UML class diagrams, state machines, and OCL specifications.

In this chapter we describe a compiler that implements a translation of a well-defined subset of UML diagrams which is sufficient for many applications. This subset consists of class diagrams which only have associations with a multiplicity of 0 or 1 and no generic classes, flat state-machines (state-machines can always be represented as a flat state machine with the same behaviour, as described by Varro in [150]), and OCL constraints.

We focus on deductive verification in higher-order logic. This allows the verification of possibly infinite-state systems. Therefore, we describe a translation of a subset of the UML into the input language of the interactive theorem prover PVS [121]. This enables the verification of the specification, originally given in OCL, using PVS.

OCL, a three-valued logic, has then to be encoded in PVS, which is based on a two-valued logic and which only allows total functions. The reason for OCL's three-

valuedness is that it uses partial functions and that relations are interpreted as strict functions into the three truth-values. Furthermore, the additional truth value only occurs by applying a function to a value outside its domain, but it is not represented by a literal.

The way partial functions of OCL are translated to PVS decides how the three-valued logic is handled. Our transformation restricts a partial function, as they occur in OCL specifications, to its domain, yielding a total function. Note that because most functions defined in specifications do not include recursive calls, the restricted domain can be easily computed. Recursive functions, which occur by way of translating recursive definitions in OCL, have to be modified by the user anyway, because in general we cannot prove that these definitions are defined for any input.

Then PVS generates proof obligations, so called *type consistency constraints* (TCC), which establish the type correctness of a PVS expression, in this case, that the function is defined for the declared domain, by way of a proof. If this fails, the user can always correct the generated output or change his constraints.[1]

This chapter is structured as follows. In Section 4.2 we describe the overall approaches of embedding a notation into a theorem prover and motivate our choice of embedding. In Section 4.3 we give a short introduction into the PVS specification language. In Section 4.4 we introduce an example used to illustrate how the translator is working. In Section 4.5 we describe the method used to formalise the input language in PVS and how our two-valued semantics of OCL is obtained. In Section 4.6 we argue that the translation is sound. Finally, in Section 4.7 we report on our initial experience with the described tool, draw conclusions, and report on related work.

4.2 Shallow versus Deep Embedding

If one translates a UML model with its OCL constraints into the logic of a theorem prover, one has to solve the problem how the different languages of the model have to be represented in this logic. For each of the languages used in this chapter the first question is whether a shallow or a deep embedding of this language is to be used.

For a *deep embedding* the abstract syntax of the language is represented as a data type in the logic of the theorem prover and a semantic function interpreting all possible terms of the language is to be provided.

The advantage of a deep embedding are:

- Meta-theorems of the language can be formulated and verified with the theorem prover, because the semantics has already been formalised in the logic of the theorem prover.

[1] This is most of the time the better solution, because our experience suggest that in this case the specification contains an error.

- The translation into the language of the theorem prover is almost trivial and very easy to check, because only the abstract syntax tree (already present in the translator) has to be written in the logic of the theorem prover as an instance of the data type.

The disadvantage of a deep embedding are:

- Depending on the complexity of the semantics, reasoning can be very complex. Many properties of the language, which are not expressible in the type system of the theorem-prover's logic have to be proved for each specification, for example, type checking of the embedded program.

- If a language like OCL is to be embedded, part of the derivation rules and decision procedures of the theorem prover may have to be duplicated in the logic of the theorem prover as a theory, in order to be applicable to the embedded specification.

In case of *shallow embedding* the language is directly represented in the logic of the theorem prover. This requires translating the language into a *verification condition* encoding the semantics of the model. If this verification condition implies the specification, then the model is assumed to satisfy the specification.

The advantages of shallow embedding are:

- It is much simpler to manipulate shallowly-embedded formulae in PVS and one can use the highly automated rules provided by the theorem prover.

The disadvantages of shallow embedding are:

- The generation of the verification conditions has to be verified. If the translation is sufficiently simple, such a proof can be established, but in other cases one has to trust the verification condition generator.

Our translation is based on a hybrid approach. UML state machines and class diagrams are embedded deeply into the theorem prover, whereas OCL is embedded shallowly.

The syntax of UML state machines is considerably simpler than its semantics. In order to trust the embedding of state machines we need to be able to prove some properties of the semantics required by the standard. Furthermore, instead of defining a single semantics for state machines, the standard defines a collection of semantics through *semantic variation points*. These variation points are made explicit in the deep embedding, where an instance of the semantics can be obtained by, for example, instantiating functions for selecting the way events are processed.

4.3 PVS Language

PVS [121] is a proof assistant, proof checker, and a language for specifications in classical higher-order logic. Higher-order logic is basically a typed lambda-calculus enriched with the two logical operators $(\forall x : T)P(x)$ describing universal quantification over elements of type T, and implication ($\Longrightarrow$). One base type, boolean, of the underlying type theory is singled out in order to define predicates. See, for example, Leivant [93] for further a more detailed description of higher-order logic. In this section we describe the essential part of the PVS specification language used by our translator.

As higher-order logic is a typed language, so is PVS. A type is either one of the elementary types predefined by PVS, like boolean or integer, or it is a user-defined uninterpreted type. For example T : TYPE declares a new uninterpreted type T in PVS. The declaration T : TYPE+ declares a uninterpreted type which contains at least one element. Interpreted types can be defined by enumerating their member, which are uninterpreted constants of this type, for example: T : TYPE = $\{a, b, c\}$.

From the base types new types may be constructed. The most important construction is a function. Predicates and sets are defined by their characteristic functions, for example, a set of elements of types T has the type setof : TYPE = $[T \rightarrow \text{bool}]$.

Of similar importance are *predicate subtypes*. Predicate subtypes allow the definition of a new type by constraining an existing type using a predicate. The syntax of the definition of a predicate subtype is: S : TYPE = $\{x : T \mid P(x)\}$. This definition defines the type S as the subtype of T such that all individuals of S satisfy the predicate P.

Other type declarations we use are structures, which are declared by T : TYPE = $[\# f_1 : T_1, \ldots, f_n : T_n \#]$, where, for $1 \leq i \leq n$, the f_i are *field names*, which are used later to select values from a structure value, and the T_i specify the type of each field. For example, if e is some PVS expression whose type is T, then the value of f_1 in e is characterised by the expression $e`f_1$.

PVS allows the user to define new constants using a similar syntax as used for defining types. For example, a new constant which is a predicate is defined by, $P(x : T)$: bool = $Q(x)$. Alternatively, the same predicate can be defined using lambda-abstractions, for example, $P : [T \rightarrow \text{bool}]$ = LAMBDA $(x : T) : Q(x)$. Constants need not be interpreted, as in the preceding examples. Uninterpreted constants, for example, arbitrary predicates on T, are introduced by $C : [T \rightarrow \text{bool}]$.

Any predicate P can be used as a type by writing it as (P), which represents the set of values satisfying the predicate. This is often used in PVS specifications: for example, injective?, which is true if and only if its argument is an injective function, is used as a type: (injective?$[T, T]$) is the type of *injective function from T into T*.

In PVS all definitions have to be either constants or total functions. This is guaranteed by the type system of PVS, where the domain and the range of functions can be precisely described. For each function application the type checker may generate TCCs which prove that the arguments are inside the domain of the function. For recursive function definitions proof obligations are generated to prove termination, which ensure

that the function is total. For example, OCL's iterate expression (see Section 2.3) over a set can be defined in PVS as the following expression explained below:

```
iterate(s : finite_set[S], a : T, f : [S, T → T]) : RECURSIVE T =
    IF empty?(s)
    THEN a
    ELSE LET x = choose(s) IN iterate(remove(x, s), f(x, a), f)
    ENDIF
  MEASURE s BY strict_subset?
```
(4.1)

If the set to iterate over is empty, then the function returns the accumulator variable a. Otherwise, an element x of the input set s is chosen using Hilbert's epsilon function [15], the iterate expression f is applied to the current accumulator value a and the chosen element x to produce a new accumulator value, and the iterate function is recursively invoked with the new accumulator value, the iterate expression f, and $s \setminus \{x\}$ (written in PVS as "remove(x, s)") as the new set to iterate over. The MEASURE definition states that the argument s has to be measured by the strict subset relation $\subset$ and is used to generate the following proof obligations:

- The values of s are strictly ordered by $\subset$ and $(2^S, \subset)$ is a well-founded set.[2]
- With each recursive step the value of s is decreasing, that is: $s \setminus \{x\} \subset s$.

These conditions together imply that *iterate* is well-defined for all finite sets.

PVS specifications are organised into parameterised theories that may contain assumptions, definitions, axioms, and theorems. Such a theory is declared in PVS as, for example:

```
statemachine[Attribute, Class, Loc, Ref, Signal: TYPE+]: THEORY
  BEGIN
    ...
  END statemachine
```

More information about PVS and its language can be found in [138] and in [124].

4.4 Running Example

We use the *Sieve of Eratosthenes* as a running example. It is modelled using the two classes *Generator* and *Sieve* (see Figure 4.1). Exactly one instance, the root object, of the class Generator is present in the model. The generator creates an instance of the Sieve class. Then it sends the new instance natural numbers in increasing order, see Figure 4.2. The association from Generator to Sieve is called *itsSieve*.

[2] In PVS a relation $<$ on a type T is well founded, if $\forall P : P \subseteq T \implies (\exists (y : y \in P) \implies (\exists y : y \in P \wedge (\forall x : x \in P \implies \neg x < y)))$.

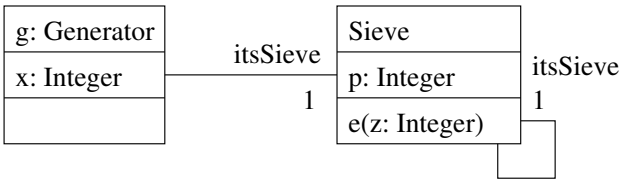


Figure 4.1: Class diagram of the Sieve example

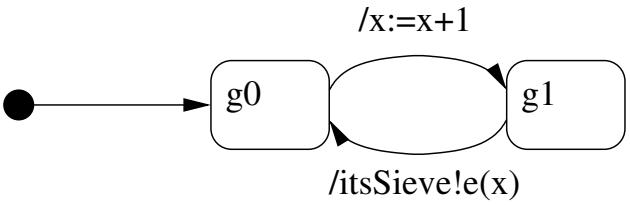


Figure 4.2: State machine of the Generator

Upon creation, each instance of Sieve receives a prime number and stores it in its attribute p. Then it creates a successor object, called *itsSieve*, and starts receiving a sequence of integers i. If p divides i, then this instance does nothing. Otherwise, it sends i to *itsSieve*. This behaviour is shown in Figure 4.3.

The safety property we would like to prove is that p is a prime number for each instance of Sieve; this can be formalised in OCL by:

$$\textbf{context } \mathit{Sieve} \textbf{ inv} : \mathit{Integer}\{2..(p-1)\} \rightarrow \mathit{forAll}(i \mid p.\mathit{mod}(i) \neq 0)$$

This constraint states that the value of the attribute p is not divisible by any number i between 2 and $p-1$. To prove this, we need to establish, that the sequence of integers received by each instance of Sieve is monotonically increasing.

We have chosen this example, because it is short, but still challenging to verify. It involves unbounded object creation and asynchronous communication, and therefore

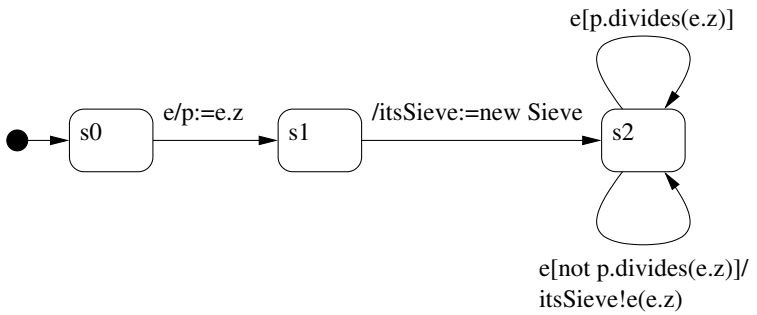


Figure 4.3: State machine of a Sieve

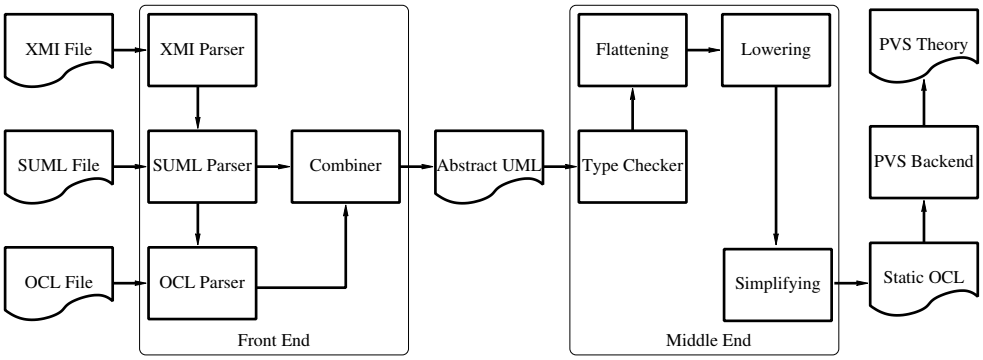


Figure 4.4: Architecture of the translator

does not have a finite state space. Furthermore, the behaviour of the model depends on the data sent between objects. Note also that the property we want to prove about the model is a number-theoretic property, namely, that the numbers generated are primes. This makes it impossible to show the considered property using automatic techniques like model checking.

4.5 Definition of the Translator

We define the translator from a subset of UML defined in this thesis into the input language of the theorem prover PVS. The restriction is, that we only allow associations in which an object is associated to at most one other object under the same association end name. Consequently, we may assume that Lemma 2.12 holds in this chapter.

The architecture of the translator is similar to that of a modern optimising compiler, using a front end for parsing input files, a middle end for analysing and transforming the parsed program into an equivalent, but optimised one, and a back end, which transforms the optimised tree into the target language (see the book of Aho, Sethi, and Ullman on compiler construction for details [4]). Here we describe the analysis and the transformations necessary for translating a UML model and its OCL requirements into the PVS language.

Figure 4.4 summarises the architecture of the translator and describes the data flow between the different parts of the system. The following sections explain each part of the translator in detail.

4.5.1 Front End

The front end is responsible for parsing the compilation units. The compilation units consists of files containing OCL constraints and the UML model.

We use existing modelling tools for creating the UML models. These tools are supposed to export the model using the Extensible Meta-data Interchange format XMI, a format specified by the Object Management Group for the exchange of UML models. This XMI format is available in five versions [103, 104, 106, 109, 108]; all of them use XML [157] to encode the model (and its abstract) syntax. All of these formats are sufficiently different, such that a different parser for each version of XMI is needed.

Another difficulty of the XMI format is that the specifications define an algorithm for generating an interchange format for a meta-model[3]. The UML standard defines a meta-model for UML models which most tools support for their diagram exchange; the generated XML document is very complex and contains a lot of redundancy. Because the UML standards are ambiguous, each tool vendor implements its own variant of the language and claims to implement XMI correctly.

To avoid implementing a parser parsing all versions and variants of XMI, which is a major engineering task, we have implemented translations from some supported variants to a language we call SUML (short for Simple UML format) using XSLT [50]. XSLT is a domain-specific language for transforming XML documents into other formats, especially other XML documents.

A SUML document, which is also an XML file, describes all concepts necessary for the tool's operation using a precisely specified abstract syntax. Parsing SUML is considerably simpler than parsing XMI because this format was designed to express a concept in precisely *one* way.

The SUML parser identifies all constraints present in an XMI or SUML file (unfortunately, many tools do not support embedding OCL into a model, and therefore do not store constraints in XMI files), and passes these to the OCL parser.

The syntax of OCL constraints supported by the parser is specified in [113]. A subset of the language is also described in Chapter 2.3.2. The OCL parser is used to parse constraints from a separate OCL file and from an SUML document into abstract syntax trees. The tool uses a recursive-descent parser, because OCL's grammar is in $LL(2)$, that is, it can be parsed top-down from left-to-right with 2 tokens of look-ahead and the parser creates a leftmost derivation.

Furthermore, the OCL grammar has been extended to parse a very simple action language, that is, a language used to specify the actions a state machine performs, which uses (a subset of) OCL as its expression language, and adds statements for signal emission, operation calls, conditional actions, loops, and assignments to attributes. These actions can be used in models to define the body of methods and to define the actions performed in state machines.

The abstract syntax trees generated by the SUML parser (representing class diagrams and state machines) and by the OCL parser (representing constraints and other expressions) are then passed to the combiner, which attaches each constraint to its context. Furthermore, it reports all constraints which do not have a context defined by

[3]The meta-model is an object-oriented implementation of an abstract syntax. See Section 1.1.

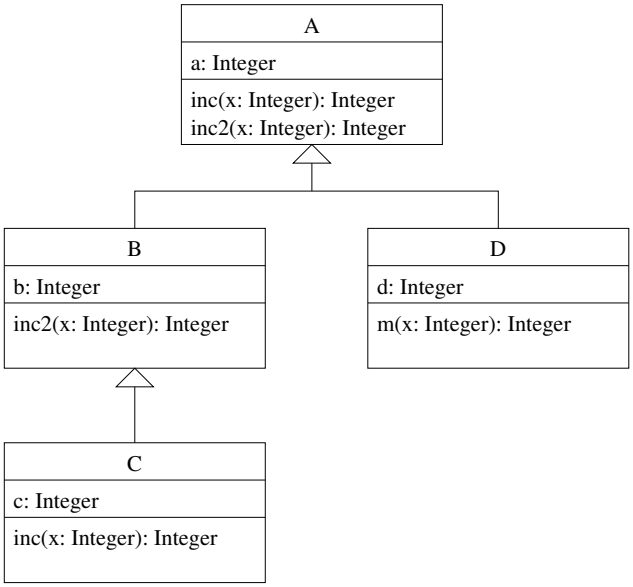


Figure 4.5: Example class diagram

a class diagram or a state machine. The result produced by the combiner is a single abstract syntax tree representing one UML model, which is passed to the middle end.

4.5.2 Middle End

The middle end is responsible for semantic analysis and semantics preserving transformations of the abstract syntax tree generated by the front end. This means, that all operations and transformations performed by the middle end are transformations of abstract syntax trees. The goal of these transformations are to transform the UML model into a form where it can be represented as a set of theories in PVS. The middle end is implemented in four phases: semantic analysis, flattening, lowering, and simplifying.

As a second running example, we use the model which consists of the class diagram in Figure 4.5.

Semantic Analysis

The first phase of the middle end consists of semantic analysis, and also contains the type checker. In this phase, the middle-end checks, whether the input model satisfies (a subset of) the well-formedness constraints specified in the standard, in particular, whether all constraints are well typed (as described in Chapter 3), annotates each use of an element (attribute, operation, and classes) in the syntax tree with its definition, whether all state machines are well-formed, and whether the class diagrams are well

formed (as described in Section 2.1). Note that this phase does not change the model or its semantics, it only annotates the model with information useful during later phases.

Furthermore, the translation may stop with reporting errors if the validation of the model fails. If validation fails, the tool has detected an inconsistency in the model, for example, a generalisation relation which is not a partial order, or one of the constraints is not well-typed.

One of these errors is the occurrence of a call to the function *oclIsUndefined*(). We have described the semantic issues with this function in Section 2.3.5, where we have explained that it is impossible to implement such a function. The behavior of this function, however, could be modeled in PVS. But this implies that the function eval has to be defined in PVS, which amounts to a deep embedding of OCL into PVS. Therefore, we have decided to remove the function from our subset of OCL.

Flattening

The second step is to flatten the model, that is, remove all uses of late binding from the model. Each call to a method is translated to a dynamic dispatch table, where the runtime-type of the callee is queried and the call of the method is statically resolved to a function. This step is necessary because PVS, as most other theorem provers, does not support late binding. Since the compiler is generating verification conditions, this transformation has to be performed by it.

Generating these calls also requires knowledge of the complete class diagram, because we need to know the overriding definitions of each operation called. If only part of the class diagram is used in this phase, then the compiler might miss an overriding definition of a method which this transformation does not take into account. The flattened tree would not call the overridden method, as required by the semantics, but one of the methods in a super-class.

For example, consider the class diagram displayed in Figure 4.5 and the OCL expression e.inc(2). The value returned by the call to inc depends on the runtime type of e, because we have a definition of this operation in classes A and C. By flattening, this dependency is made explicit by replacing the expression e.inc(2) with guarded static calls:

$$\begin{array}{l}\textbf{if } e.oclIsTypeOf(A) \textbf{ then } e.A :: \text{inc}(2)\\ \quad \textbf{else if } e.oclIsTypeOf(B) \textbf{ then } e.A :: \text{inc}(2)\\ \quad\quad \textbf{else } e.C :: \text{inc}(2) \textbf{ endif endif}\end{array}$$

Here, the notation $C :: \text{inc}(2)$ refers to a static call of inc(2) in class C and $A :: \text{inc}(2)$ to a static call of inc(2) in class A. Finally, if e is an instance of class B, then the implementation defined in class A is used, because objects of type B inherit the implementation in class A.

Lowering

The third step is to rewrite the abstract syntax tree, into one using simpler concepts. The most important part of lowering is to replace each occurrence of an, now statically determined, operation call by simpler expressions, provided that the meaning of an operation is defined using OCL.

An OCL definition can be obtained from a behavioural specification of an operation, if it is either specified using the pattern:

$$\textbf{context}\ c :: m(\vec{v}) : T\ \textbf{pre} : \text{true}\ \textbf{post} : \mathit{result} = e$$

or, more directly, using the declaration:

$$\textbf{context}\ c :: m(\vec{v}) : T\ \textbf{body} : e \quad .$$

The first pattern is quite common in the UML and in the OCL standard. If such a pattern is recognised, then the call to this operation is replaced by the expression e, where each formal parameter, which also occur in e, is replaced by the expression describing the actual arguments. This replacement is implemented as follows:

If we encounter a call $c :: m(\vec{e})$ in an OCL expression e' and we identify a definition e of $c :: m$, the call is replaced by

$$\textbf{let}\ f(\vec{v}) : T\ = e\ \textbf{in}\ f(\vec{e}) \quad ,$$

where f and $\vec{v}$ do not occur in e'. To handle the case of a recursive operation call, we replace calls to $c :: m$ in e by f, too. Observe that we have to re-introduce recursive let-definitions into OCL 2.0 [113], which were present in OCL 1.4 [105] but have been removed for the latest version of the standard.

Note that, after flattening, all calls are statically determined, so this transformation can almost always be performed. The only exception are mutually recursive definitions. In principle, mutually recursive functions can be obtained using the same mechanism in OCL, but the PVS specification language does not allow them.

Simplifying

Lowering usually introduces a lot of redundancy into the model, for example, useless cases in the dynamic dispatch tables and duplication of expressions. The simplifying pass uses algebraic laws to simplify all expressions. Therefore, the next pass, simplifying, is used to rewrite the model using algebraic simplifications.

The important part of this step is to reduce the number of cases which have to be distinguished for each call of an operation, whose semantics depends on the type of the callee.

4.5.3 Back End

The back end transforms an abstract syntax tree, which has been generated by the middle end, into the PVS specification language. The resulting theory is suitable as input for the PVS theorem prover.

One obvious shortcoming of the back end is that it cannot provide measure functions used in the type checker of PVS to verify that each function is computable and well-defined. The user may review the generated theory and provide these measures by hand-coding them into the PVS specification.

Class Diagrams

A class diagram is embedded deeply into PVS. In PVS we define a type Class which enumerates all classes occurring in the model. For example, the class diagram shown in Figure 4.1 leads to the definition:

$$\text{Class} : \text{TYPE+} = \{\text{OclAny}, \text{OclVoid}, \text{OclInvalid}, \text{Generator}, \text{Sieve}\} \quad .$$

Observe that OclAny, OclVoid, and OclInvalid are implicitly defined in the class diagram, as specified by OCL (see Section 2.3.4).

Next we define a subtype Active, which enumerates all classes occurring in Class which are active classes. In the sieve example, the classes Generator and Sieve are active, therefore, we define:

$$\text{Active} : \text{TYPE+} = \{x : \text{Class} \mid x = \text{Generator} \vee x = \text{Sieve}\} \quad .$$

The constant rootClass represents an active class from which the first object of the system is instantiated.

$$\text{rootClass} : \text{Active} = \text{Generator}$$

Furthermore, the attribute, operation, signal, and reference names of each class are enumerated as a type. Each name is prefixed with the name of the class which defines it.

Attribute : TYPE = {Generator___x, Sieve___z, Sieve___p, unusedAttribute}
Reference : TYPE = {Sieve___itsSieve, Generator___itsSieve,
Sieve___itsGenerator, unusedReference}
Operation : TYPE = {}
Signal : TYPE = {Sieve___e}

The association of a name to its defining class is done by prefixing the name with the corresponding class name. The translation of the class diagram shown in Figure 4.1 is presented in Figure 4.6.

As described in Section 2.2, objectstructures are used to interpret a class diagram. For sake of simplicity, we consider only associations where to each object at most one

```
Class: TYPE+ = { Generator, Sieve }
Active: TYPE+ = { x: Class | x=Generator OR x=Sieve }
rootClass: (active) = Generator
Attribute: TYPE+ = { Generator___x, Sieve___z, Sieve___p, unusedAttribute }
Reference: TYPE+ = { Sieve___itsSieve, Generator___itsSieve,Sieve___itsGenerator,
  unusedReference }
```

Figure 4.6: Translation of the Sieve class diagram

object can be associated. In this case, association end names can be treated as attributes of objects.

$$
\begin{array}{l}
\text{ObjectId} : \text{TYPE+} = \text{nat} \\
\textit{null} : \text{ObjectId} = 0 \\
\text{ObjectIdNotNull} : \text{TYPE+} = \{x : \text{ObjectId} \mid x \neq \textit{null}\} \\
\text{Object} : \text{TYPE} = [\#\text{class} : \text{Class}, \\
\qquad \text{aval} : [\text{Attribute} \rightarrow \text{Value}], \\
\qquad \text{rval} : [\text{Reference} \rightarrow \text{ObjectId}]\ \#] \\
\text{State} : \text{TYPE} = [\text{ObjectIdNotNull} \rightarrow \text{Object}]
\end{array}
\tag{4.2}
$$

ObjectId is the type of all object identifiers. We define *null* as an element of type ObjectId and define a subtype ObjectIdNotNull of ObjectId.[4] The type Object consists of a valuation function aval that assigns values to all attribute names, a function rval that assigns values to all reference names (or association end names), and a field class referencing the class of which the object is an instance of. Finally, a (global) state, or object diagram is defined as a function from ObjectIdNotNull to Object. Observe, that *null* does not have an object in this state. Also, whenever an object is accessed, PVS will generate type consistency constraints which require the user to prove that the object accessed is not *null*.

Type checking in the middle-end asserts that all objects only uses attributes defined for their class, so we assign to the attributes not defined in an object's class an arbitrary value. Therefore, we have deliberately simplified the back-end to generate for each object the attributes occurring in all classes of the model. A value for this attribute is specified only if the attribute occurs in the full descriptor of the class of the object.

Because PVS only allows total functions, we have an infinite number of objects in the state (it is indeed defined as a function nat → Object). In the semantics, we assign to each "slot" in the state, which represents an object which does not exists, an "instance" of OclInvalid. Using the natural numbers, we can obtain a slot for a new object by the PVS expression new(s : State) : ObjectIdNotNull = min{a : ObjectIdNotNull | class(state(a)) = OclInvalid}.

[4] Using natural numbers as object identifiers (indicated by the PVS type nat) is only a convenience. We could have left the type ObjectId uninterpreted.

State Machines

We use a hybrid approach for embedding state machines into PVS, using a deep embedding to represent the structure of the state machine and a shallow embedding of guards and expressions.

The back-end generates a type for all the locations of the state machines occurring in the model, prefixing it with the name of the class in which the state has been defined. Because states need not be named in UML and need not be unique for a state machine, we use an arbitrary number or string, which is defined in the XMI file. Therefore, the locations of the sieve example are:

$$\begin{aligned}\text{Location} : \text{TYPE} = \{&\text{Generator\_\_\_\_61}, \text{Generator\_\_\_\_64}, \text{Generator\_\_\_\_66},\\ &\text{Sieve\_\_\_\_25}, \text{Sieve\_\_\_\_28}, \text{Sieve\_\_\_\_32}, \text{Sieve\_\_\_\_33}\}\end{aligned}$$

Similar to the way active classes are represented, we define a subtype of Location containing all initial locations:

$$\text{Initial} : \text{TYPE} = \{\ell : \text{Location} \mid \ell = \text{Sieve\_\_\_\_25} \vee \ell = \text{Generator\_\_\_\_61}\}$$

Transitions are embedded deeply, but parts of it use a shallow embedding. A transition is defined by the class of the state machine, a source location, a target location, a trigger, a guard, which is any PVS function mapping a global state into the booleans, and a list of actions defined in the action language. The type of a transition is:

$$\begin{aligned}\text{Transition} : \text{TYPE} = [&\#\text{class} : \text{Class}, \text{source} : \text{Location}, \text{target} : \text{Location},\\ &\text{trigger} : \text{Event}, \text{guard} : [\text{State} \to \text{bool}], \text{action} : \text{List[Action]}, \#]\end{aligned}$$

Next, we define all transitions occurring in the state machine as constants of type Transition. Again, the names are chosen using the XMI representation, such that they are unique for the complete PVS theory. For example, the transition from *state_1* to *state_1* sending an integer to the next object, as shown in Figure 4.2, is translated to:

$$\begin{aligned}&t28 : \text{Transition} = (\#\text{source} := \text{Sieve\_\_\_\_28},\\ &\quad \text{trigger} := \text{signalEvent}(\text{Sieve\_\_\_}e, \text{Sieve\_\_\_}z),\\ &\quad \text{guard} := \lambda(\text{val} : \text{State}) : (\neg\text{divides}((\text{val`aval}(\text{Sieve\_\_\_}p)),\\ &\qquad (\text{val`aval}(\text{Sieve\_\_\_}z)))),\\ &\quad \text{actions} := (\text{cons}((\text{emitSignal}(\text{Sieve\_\_\_itsSieve}, \text{Sieve\_\_\_}e,\\ &\qquad \lambda(\text{val} : \text{State}) : (\text{val`aval}(\text{Sieve\_\_\_}z))))), \text{null}),\\ &\quad \text{target} := \text{Sieve\_\_\_\_28},\\ &\quad \text{class} := \text{Sieve}\ \#)\end{aligned}$$

Finally, the translator defines a set of all transitions occurring in the model. The result of this enumeration of transitions is shown in Figure 4.7.

The semantics of the translated state machines is given in terms of sets of computations using a PVS theory provided by Hooman and van der Zwaag and described in [67].

```
Location: TYPE+ = { Generator____61, Generator____64, Generator____66,
   Sieve____25, Sieve____28, Sieve____32, Sieve____33 }

Initial:TYPE={l: Location | l = Sieve____25 OR l = Generator____61 }

⋮

t28: Transition = (#
   source := Sieve____28,
   trigger := signalEvent(Sieve___e, Sieve___z),
   guard := (LAMBDA (val: Valuation):
      (NOT divides((val‘aval(Sieve___p)), (val‘aval(Sieve___z))))),
   actions := (cons((emitSignal(Sieve___itsSieve,
      Sieve___e, (LAMBDA (val: Valuation): (val‘aval(Sieve___z))))), null)),
   target := Sieve____28,
   class := Sieve
#);

⋮

transitions: setof[Transition] = { t: Transition | t = t9 OR t = t11 OR
   t = t13 OR t = t25 OR t = t28 OR t = t32 OR t = t34 OR t = t37 }
```

Figure 4.7: Translation of the Generator state machine

OCL

Finally, we have to generate a specification for OCL expressions and actions. This involves "embedding" the three-valued logic of OCL (see Section 2.3.5) into the two-valued logic of PVS .

For this embedding, we use the fact that the OCL standard requires that all constraints have to be true for a well-formed model. Because we are interested in verifying the *correctness* of a model with respect to its specification, we actually have to prove that all OCL constraints are *true*, that is, neither *false* nor *undefined*. If an expression evaluates to undefined, then the constraint containing it should not be provable.

We do this by using a shallow embedding, such that all constraints that evaluate to true also evaluate to true in PVS and all constraints which evaluate to false also evaluate to false in PVS. The translation was defined in a way that constraints which evaluate to undefined ($\mathcal{J}$) using the evaluation function defined in Definition 2.19 lead to unprovable type consistency constraints. Consequently, models containing constraints with subexpressions that evaluate to undefined are not provable in PVS.

1. We do not need to require that each function is strict in its arguments. Instead, we use the requirement that each function application yields a well-defined value. PVS implements this requirement by generating type consistency constraints automatically, in order to require a proof that each argument supplied is in the domain of the function and that recursively defined functions always have a defined value for each of its arguments.

2. We do not have to redefine the core logic, for example, the *and* and *implies* functions, to properly define the functions described in Table 2.2, in PVS. This entails the development of specialised strategies to automate reasoning in a three-valued logic. Using our approach we benefit from PVS's high degree of automation.

3. We removed the function *oclIsUndefined* from OCL, that is, the tool does not translate models which contain constraints using this function. For this function we were not able to give a definition without implementing a deep embedding and we want to avoid a deep embedding. This requires that the user rewrites the constraints in such a way that he does not need to use *oclIsUndefined*. This can be done by identifying the condition φ under which constraint ψ may be undefined and writing a corresponding constraint $\neg\varphi \wedge \psi$, if one expects the constraint to be false if it is undefined, or $\neg\varphi \implies \psi$ if he does not care about these undefined cases.

Primitive functions. Some primitive functions used in OCL are partial functions which may result in undefined, for example, division if the dividend is zero. In [20] Brucker and Wolff have extended the partial functions to total functions by explicitly introducing undefined and formalising the underlying three-valued logic. This

approach is a deep embedding, which enabled them to prove certain meta-theoretic properties about different embeddings in OCL. The main drawback for verifying a concrete system is that by reasoning in a three-valued logic one loses PVS's high degree of automation. The main reason is, that for OCL the law of the excluded middle ($\vdash p \vee \neg p$) and the axiom $\vdash p \implies p$ do not hold. However, in PVS it is assumed that these axioms always hold in its strategies.

Instead, we *restrict* each partial function to its domain making it a total function. This requires formalising the domains of each primitive function defined in OCL. Most functions of the OCL standard library already have an equivalent definition in PVS, for example, the arithmetic functions. We have defined a library of total functions which are not provided by PVS.

A consequence of this encoding is, that whenever an expression in OCL may evaluate to undefined, the corresponding expression in PVS has an unprovable type consistency constraint. Conversely, if all type consistency constraints of the translated PVS expression are provable, the original OCL expression never evaluates to undefined!

Null references. OCL allows user-defined functions in expressions, which are either introduced using a let construct or by using an operation that has been declared side-effect free. Such functions have to be represented in PVS. We can define such a definition if the meaning of the function is given as an OCL expression or if its semantics has been defined using PVS expressions.

The signature of such a function is obtained from the type definitions computed from the class hierarchy. All instances in a model are identified by a value of the type *OclAny*, which is represented in PVS by the type ObjectId. This type contains a special object *null* which represents the non-existing object. We refine these types to reflect the class hierarchy using predicate subtypes. First the back-end generates a partial order $\preceq$ which encodes the generalisation hierarchy of the class diagram. For the Sieve example, this predicate is:

$$\preceq : (\text{partial\_order?}[\text{Class}]) = \lambda(x, y : \text{Class}) : (x = y) \vee (x = \text{Generator} \wedge y = \text{OclAny}) \vee (x = \text{Sieve} \wedge y = \text{OclAny}) \quad .$$

Then for each class C defined in the class diagram a predicate subtype

$$C\text{IdNotNull} = \{x : \text{ObjectIdNotNull} \mid x\text{`class} \preceq C\} \quad .$$

is defined. This encodes the usual the subsumption property that if a class C is a subclass of D, then each instance of C is also an instance of D. Because parameters may be *null*, we define

$$C\text{Id} = \{x : \text{ObjectId} \mid x = \mathit{null} \vee x\text{`class} \preceq C\} \quad .$$

The signature of a user-defined method $C :: m(v_1 : T_1, v_2 : T_2, \ldots, v_n : T_n) : T$ will be defined as:

$$C\_\_\_m : [C\text{IdNotNull}, T_1\text{Id}, T_2\text{Id}, \ldots, T_n\text{Id} \rightarrow T\text{Id}]$$

and a corresponding function encoding the semantics of the OCL function as defined below.

Undefined values, which occur, for example, by accessing attributes of *null* or array members outside of the bounds of the array, or retyping (downward-casting) an object to a subtype of its real type, are avoided by the formalisation of signatures. For example, requiring that the first parameter of a PVS function, which is used for the *self* reference, is non-null causes the proof checker to generate a TCC, which establishes that the value of *self* is never *Null*. Furthermore, type checking guarantees that for every object the value of each of its attributes is defined. This property is maintained by the behavioural semantics.

Recursion. The formal semantics of OCL is concerned with *executing* OCL constraints. Consequently, the value of an recursive function is undefined, if its evaluation is diverging. These problems do not occur in PVS, because it is not possible to define a diverging recursive function in PVS. For each recursive function definition a ranking function has to be supplied, which is used in a termination proof. We translate recursive functions of OCL to recursive functions in PVS directly and ask the user to supply a suitable ranking function in the PVS output, because it is not possible to automatically compute such a function.

Example 4.1. In OCL one might define Fibonacci's function recursively as:

> **context** *Integer* :: fib() : *Integer*
> **pre** : *self* ≥ 0
> **body** : **if** *self* > 1 **then** (*self* $- 2$).fib() + (*self* $- 1$).fib() **else** 1 **endif**

Assuming that *Integer* is not sub-classed, this expression is translated to PVS as:

```
Integer___fib(self : int | x ≥ 0) : RECURSIVE int =
    IF self > 1 THEN Integer___fib(self − 2) + Integer___fib(self − 1) ELSE 1 ENDIF
  MEASURE ...
```

The measure function is not generated, because it cannot be generally computed automatically. ♦

As an alternative we propose to add a ranking function on the level of OCL, for example, using a stereotype **rank** mapping the arguments of the recursive functions into the non-negative integers.

Example 4.2. Extending the definition of Fibonacci's function with a rank we obtain:

> **context** *Integer* :: fib() : *Integer*
> **pre** : *self* ≥ 0
> **body** : **if** *self* > 1 **then** (*self* − 2).fib() + (*self* − 1).fib() **else** 1 **endif**
> **rank** : *self*

Now this expression is translated to PVS as:

```
Integer___fib(self : int | x ≥ 0) : RECURSIVE int =
    IF self > 1 THEN Integer___fib(self − 2) + Integer___fib(self − 1) ELSE 1 ENDIF
  MEASURE self
```

In this case a number of type consistency constrains are generated which establish that the expression is well-typed and that the function is totally defined. For example, one of the termination constraints, which establishes that the rank is decreasing, is: self > 1 ⟹ self − 2 < self. ♦

Definition of the Translation. Having given an overview of the central ideas of the translation, we formally define the translation function *trans* in this section and comment on each decision.

Definition 4.1 (Translation of Types)**.** The translation of types is defined by:

1. $trans(Boolean) \stackrel{\text{def}}{=} \text{bool}$
2. $trans(Integer) \stackrel{\text{def}}{=} \text{int}$
3. $trans(Real) \stackrel{\text{def}}{=} \text{real}$
4. $trans(Bag(T)) \stackrel{\text{def}}{=} \text{finite\_bag}[trans(T)]$
5. $trans(Set(T)) \stackrel{\text{def}}{=} \text{finite\_set}[trans(T)]$
6. $trans(Sequence(T)) \stackrel{\text{def}}{=} \text{finseq}[trans(T)]$
7. $trans(C) \stackrel{\text{def}}{=} C\text{Id}$ ◊

Definition 4.2 (Translation of Expressions)**.** Let $\mathcal{V}$ be a set of names we call *variables*. Let D be a class diagram. In the following, state and prestate are two free variables. The resulting PVS expression will then be bound by lambda abstractions to obtain a function on actions to booleans.

1. *trans*(true) = true, *trans*(false) = false, and *trans*(null) = *null*.
2. For each number n define $trans(n) = n$.

3. For collection expressions, we show the case of representing a sequence, the other collections are analogous: Let $e_1, e_2, \ldots, e_n$ be a list of OCL expressions, which may be empty. Then

$$trans(Sequence\{e_1, e_2, \ldots, e_n\}) = \text{cons}(trans(e_1), trans(Sequence\{e_2, \ldots, e_n\}))$$

and

$$trans(Sequence\{\}) = \text{null} \quad .$$

For a range expression $e..e'$ define $trans(e..e') = \text{range}(T(e), T(e'))$, where range is defined in a library as:

$$\begin{array}{l} \text{range}(x : \text{int}, y : \text{int}) : \textsc{recursive}\ \text{list}[int] = \\ \quad \textsc{if}\ x < y \\ \quad \textsc{then}\ \text{cons}(x, \text{range}(x + 1, y)) \\ \quad \textsc{else}\ \text{null} \\ \quad \textsc{endif} \\ \textsc{measure}\ y - x \end{array}$$

4. For any variable $v \in \mathcal{V}$ define $trans(v) = v$.

5. For conditional expressions define

$$trans(\textbf{if}\ t\ \textbf{then}\ t_1\ \textbf{else}\ t_2\ \textbf{endif}) = \textsc{if}\ trans(t)\ \textsc{then}\ trans(t_1)\ \textsc{else}\ trans(t_2)\ \textsc{endif} \quad .$$

6. For let-expressions define

$$\begin{array}{l} trans(\textbf{let}\ v_0(\vec{v_0}) : T_0 = t_0, \ldots, v_n(\vec{v_n}) : T_n = t_n\ \textbf{in}\ t) = \\ \quad \textsc{let}\ trans(v_0)(trans(\vec{v_0})) : trans(T_0) = trans(t_0), \\ \qquad \ldots \\ \qquad trans(v_n)(trans(\vec{v_n})) : trans(T_n) = trans(t_n) \\ \quad \textsc{in}\ trans(t) \quad . \end{array}$$

7. For attribute call expressions define

$$trans(t.a) = \text{state}(trans(t))\text{`aval}(\text{pvsattrname}(t, a)) \quad ,$$

where pvsattrname obtains the type of t deduced by the type checker, looks up the class in which a has been defined and returns the name defined for this attribute in the PVS formalisation of PVS, as described above. Recall, that the member aval of the structure of type Object refers to the valuation of an objects attribute.

8. For previous attribute call expressions define

$$trans(t.a@\mathbf{pre}) = \text{prestate}(trans(t))\text{`aval}(\text{pvsattrname}(t, a)) \quad .$$

9. For operation call expressions, where the type of t_0 is not a collection type, define

$$trans(t_0.m(t_1, t_2, \ldots, t_n)) = \\ \text{pvsopername}(t_0, m)(trans(t_0), trans(t_1), trans(t_2), \ldots, trans(t_n)) \quad ,$$

where pvsopername(t_0, m) expands to the name of a function defined in the PVS library or computed from the model, which implements the semantics of m in PVS.[5]

For operation call expressions, where the type of t_0 is of type sequence, define:

$$trans(t_0.m(t_1, t_2, \ldots, t_n)) = \text{map}(trans(t_0), \\ \text{LAMBDA } x : \text{pvsopername}(t_0, m)(x, trans(t_1), trans(t_2), \ldots, trans(t_n))) \quad .$$

The other collection types use analogous definitions.

10. For collection property call expressions define:

$$trans(t_0 \rightarrow m(t_1, t_2, \ldots, t_n)) = \\ \text{pvsopername}(t_0, m)(trans(t_0), trans(t_1), trans(t_2), \ldots, trans(t_n)) \quad ,$$

where pvsopername(t_0, m) expands to the name of a function defined in the PVS library defining the semantics of the call.

11. For an iterate expressions define

$$trans(t_0 \rightarrow iterate(v_0 : T_0; a : T = t \mid t_n)) = \\ \text{iterate}(trans(t_0), trans(t), \\ \text{LAMBDA } (v_0 : trans(T_0), a : trans(T)) : \ trans(t_n))$$

For quantification expressions, however, we define:

a) $trans(t_0 \rightarrow forAll(v_0 : T_0 \mid t_n)) = \text{FORALL } (v_0 : (trans(t_0))) : trans(t_n)$

b) $trans(t_0 \rightarrow exists(v_0 : T_0 \mid t_n)) = \text{EXISTS } (v_0 : (trans(t_0))) : trans(t_n)$

[5] The definition of pvsopername(t, m) for operations and types defined in the OCL standard library is described in Appendix B.

12. For a flatten expression define

$$trans(t \rightarrow flatten()) = \text{flatten}(trans(t)) \quad ,$$

if the type of t is a collection of collections, and, otherwise, define:

$$trans(t \rightarrow flatten()) = trans(t) \quad .$$

13. For all instances of T expressions define

$$trans(T.allInstances()) = \{x : \text{ObjectIdNotNull} \mid \text{state}(x)\text{`class} = trans(T)\}$$

if T is an object type, and

$$trans(T.allInstances()) = \{x : trans(T) \mid \textsc{True}\} \quad ,$$

otherwise. For the @**pre** modified expression, replace state with prestate in the above translations. ◊

Looking at the definition of the translation function, it becomes apparent that the generated PVS functions have a structure very similar to the original OCL constraint, while the generated PVS specification does not embed a three-valued logic. The advantage of this approach is that proofs of the verification conditions are easier to find.

4.6 Soundness of the Translation

We establish the soundness of our translation. By soundness we mean that for all OCL constraint φ, if their translation is provable, then it evaluates to true in any model. For the proofs in this section, we assume the following:

Assumption 4.3. The logic used by PVS and its implementation is sound.

The logic used by PVS is well-understood and sound (see Owre and Shankar [123] for references). Whether the implementation and the "oracles", that is, automatic decision procedures like model checkers, used by PVS are sound is unknown, because the PVS system has not yet been formally verified. Therefore we have to assume the soundness of the system.

The first step of the soundness proof is relating the carriers of types, described by a function $\mathfrak{D} : \mathcal{T} \rightarrow U_\omega$, in OCL, as defined in Definition 2.18, to their translated types in PVS, as obtained by Definition 4.1. The carriers of the translated types in PVS are obtained from the carriers of the original types in OCL by removing the values $\mathcal{J}$ from the carriers in OCL , as expressed by the following proposition:

Proposition 4.4. *For the translation of types the following properties hold:*

1. $\mathfrak{D}_{\mathrm{OCL}}(\mathit{Boolean}) \setminus \{\mathcal{J}_{\mathbb{B}}\} = \mathfrak{D}_{\mathrm{PVS}}(\mathrm{bool})$.
2. $\mathfrak{D}_{\mathrm{OCL}}(\mathit{Integer}) \setminus \{\mathcal{J}_{\mathbb{R}}\} = \mathfrak{D}_{\mathrm{PVS}}(\mathrm{int})$.
3. $\mathfrak{D}_{\mathrm{OCL}}(\mathit{Real}) \setminus \{\mathcal{J}_{\mathbb{R}}\} = \mathfrak{D}_{\mathrm{PVS}}(\mathrm{real})$.
4. *For all types* τ *it holds* $\mathfrak{D}_{\mathrm{OCL}}(\mathit{Bag}(\tau)) \setminus \{\mathcal{J}_{\mathbb{N}^{\mathfrak{D}(\tau)}}\} = \mathfrak{D}_{\mathrm{PVS}}(\mathrm{finite\_bag}[\mathit{trans}(\tau)])$.
5. *For all types* τ *it holds* $\mathfrak{D}_{\mathrm{OCL}}(\mathit{Set}(\tau)) \setminus \{\mathcal{J}_{2^{\mathfrak{D}(\tau)}}\} = \mathfrak{D}_{\mathrm{PVS}}(\mathrm{finite\_set}[\mathit{trans}(\tau)])$.
6. *For all types* τ *it holds* $\mathfrak{D}_{\mathrm{OCL}}(\mathit{Sequence}(\tau)) \setminus \{\mathcal{J}_{\mathfrak{D}(\tau)^{\mathbb{N}}}\} = \mathfrak{D}_{\mathrm{PVS}}(\mathrm{finseq}[\mathit{trans}(\tau)])$.
7. *For all object types* τ *it holds:* $\mathfrak{D}_{\mathrm{OCL}}(\tau) \setminus \{\mathcal{J}_{\mathbb{O}}\} = \mathfrak{D}_{\mathrm{PVS}}(\tau\mathrm{Id})$. *Recall that object types are of kind* $\star$.

Proof. The proof is by induction on the construction of types. □

From now on we do not distinguish the carriers of types of OCL from the ones of PVS, because they have the same semantics, only the names of the types differ.

Knowing that the carriers of the types agree in PVS and OCL we lift this property to the formalisation of object diagrams, as defined in Definition 2.10, in PVS, which is defined in Equation (4.2). This is expressed in the next lemma.

Lemma 4.5. *For all object diagrams* $\sigma \in \Sigma$ *there exists a* $v \in \mathfrak{D}(\mathrm{State})$ *such that there exists an isomorphism between* v *and* σ*, written* $v \simeq \sigma$.[6]

Proof. Let Σ be the set of all object diagrams and $\sigma = \langle \tau, \xi, [\![\cdot]\!] \rangle \in \Sigma$ an object diagram.

By definition of $\mathbb{O}$ as a countable enumerable set, we find a bijective function $f_{\mathbb{O}}$ between $\mathbb{O}$ and $\mathbb{N}$ such that $f_{\mathbb{O}}(\mathit{null}) = 0$.

Let o be an object. An object state is defined in PVS as: (# class := $\tau(o)$, aval := $\xi(o)$, rval := $\lambda e : \iota x : o \overset{e}{\twoheadrightarrow} x$ #) (recall, that in the setting of this chapter encoding associations as attributes named by the association end names is justified by Lemma 2.12). For each object identifier o which is not defined in τ define the object state in PVS by (# class := OclInvalid, aval := $\lambda n : 0$, rval := $\lambda n : \mathit{null}$ #). We say that an object state ω is of the same form as its object o in PVS, written $\omega \simeq o$, if the one can be obtained from the other using the above construction.

Finally, define the instance s of State by point-wise extension, such that for all $o \in \mathbb{O}$ we have $s(f_{\mathbb{O}}(o)) \simeq \sigma(o)$. □

Next, we consider the valuation of freely occurring local variables. Assume that the translation of a well-typed OCL formula is well-typed. Then the translation of variables in Definition 4.2, Item 4 indicates that we may use the local valuations η, which interpret the OCL constraint, for interpreting the translated formula in PVS, too.

[6] See Equation (4.2) for the definition of State.

In PVS we do not have a value for undefined. One only has a proof of a property if all TCCs generated for a theory have been proved. Only if these type consistency constraints have been proved, a theory in PVS is considered well-typed. Consequently, we have to prove, that if a term is well-typed in PVS, then the value of the corresponding OCL constraint is not undefined. This is established by the following lemma:

Lemma 4.6. *Let φ be a well-typed OCL constraint such that trans(φ) is well-typed in PVS, which implies that all type consistency constraints are provable. Then for all $\overleftarrow{\sigma}, \sigma$ and valuation η of the local variables the equation* $\text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(\varphi) = [\![\mathit{trans}(\varphi)]\!]\eta$ *holds.*

Proof. Let φ be an OCL constraint. The proof is by structural induction on the construction of φ, showing that for each application of $\text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)$ there is a corresponding step in PVS.

Let $\overleftarrow{s}$ and s be states in PVS such that $\overleftarrow{s} \simeq \overleftarrow{\sigma}$ and $s \simeq \sigma$, where $\simeq$ is defined as in the proof of Lemma 4.5.

1. If $\mathit{trans}(\varphi) = \textsc{true}$, then $\varphi = \text{true}$, and consequently $\text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(\text{true}) = [\![\textsc{true}]\!] = \text{true}$.

   If $\mathit{trans}(\varphi) = \textsc{false}$, then $\varphi = \text{false}$, and consequently $\text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(\text{false}) = [\![\textsc{false}]\!] = \text{false}$.

   If $\mathit{trans}(\varphi) = \textsc{null}$, then $\varphi = \text{null}$, and, as a consequence, $\text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(\text{null}) = [\![\textsc{null}]\!] = 0$.

2. For numeric literals ℓ it holds that $\mathit{trans}(\ell) = \ell$, and, as a consequence, in all environments η $\text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(\ell) = [\![\ell]\!]$ holds.

3. Let $e_1, e_2, \ldots, e_n$ be a list of OCL expressions, which may be empty (in which case $n = 0$), such that $\text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(e_i) = [\![\mathit{trans}(e_i)]\!]\eta$ holds for all $1 \leq i \leq n$ as an induction hypothesis. We prove the case of $\mathit{Sequence}\{e_1, e_2, \ldots, e_n\}$ by induction on i.

   For empty sequences, which satisfy $i = 0$, $\mathit{trans}(\mathit{Sequence}\{\}) = \mathit{null}$ holds and both $\text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(\mathit{Sequence}\{\})$ and $[\![\mathit{null}]\!]$ result in the empty sequence.

   For singleton sequences, which satisfy $i = 1$,

   $$\mathit{trans}(\mathit{Sequence}\{e_1\}) = \text{cons}(\mathit{trans}(e_1), \mathit{null})$$

   holds and, by induction hypothesis $\text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(e_1) = [\![\mathit{trans}(e_1)]\!]\eta$, we conclude that

   $$\text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(\mathit{Sequence}\{e_1\}) = [\![\text{cons}(\mathit{trans}(e_1), \mathit{null})]\!]\eta \quad .$$

Finally, assume $i > 1$ and, as additional induction hypothesis:

$$\text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(\mathit{Sequence}\{e_1, e_2, \ldots, e_{i-1}\}) = \\ [\![\mathit{trans}(\mathit{Sequence}\{e_1, e_2, \ldots, e_{i-1}\})]\!]\eta \quad . \tag{4.3}$$

By observing that

$$\text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(\mathit{Sequence}\{e_1, e_2, \ldots, e_i\}) = \\ \text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(\mathit{Sequence}\{e_1, e_2, \ldots, e_{i-1}\} \rightarrow \mathit{append}(e_i))$$

and by using the semantics of the *append* method, we conclude

$$\text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(\mathit{Sequence}\{e_1, e_2, \ldots, e_i\}) = \\ \text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(\mathit{Sequence}\{e_1, e_2, \ldots, e_{i-1}\}) \cdot \text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(e_i) \quad .$$

Next, we use the fact, that cons describes $\cdot$ in PVS, such that

$$[\![\mathit{trans}(\mathit{Sequence}\{e_1, e_2, \ldots, e_i\})]\!]\eta = \\ [\![\mathit{trans}(\mathit{Sequence}\{e_1, e_2, \ldots, e_{i-1}\} \rightarrow \mathit{append}(e_i))]\!]\eta$$

Using the induction hypothesis (4.3) and $\text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(e_i) = [\![\mathit{trans}(e_i)]\!]\eta$ we conclude that for all $0 \leq i \leq n$ we have

$$\text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(\mathit{Sequence}\{e_1, e_2, \ldots, e_i\}) = [\![\mathit{trans}(\mathit{Sequence}\{e_1, e_2, \ldots, e_i\})]\!]\eta \quad ,$$

which, for the case $i = n$ establishes the claim for this case.

It remains, that $\text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(e..e') = [\![\mathit{trans}(e..e')]\!]\eta$ holds for range expressions $e..e'$. This follows the proof of this case applied to Definition 2.19, Item 3c, and Definition 4.2.

4. If v is a variable expression, then $\mathit{trans}(v) = v$, and, as a consequence:

$$\text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(v) = \eta(v) = [\![v]\!]\eta \quad .$$

5. Let t', t'', t''' be OCL expressions and $t =$ **if** t' **then** t'' **else** t'''**endif**.

   By Definition 2.19, Item 5 we have:

$$\mathit{trans}(t) = \text{IF } \mathit{trans}(t') \text{ THEN } \mathit{trans}(t'') \text{ ELSE } \mathit{trans}(t''') \text{ ENDIF} \quad .$$

   Then $\mathit{trans}(t)$ is, by assumption, well typed, $\mathit{trans}(t')$ has type bool and $\mathit{trans}(t'')$ and $\mathit{trans}(t''')$ are expressions of the same type in PVS, say τ.

Because $trans(t)$ is well-typed in PVS, we may conclude from the induction hypothesis:

$$\text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(t') = [\![trans(t')]\!]\eta \quad .$$

If $[\![trans(t')]\!]\eta = \text{true}$, then $[\![trans(t)]\!]\eta = [\![trans(t'')]\!]\eta$ by the semantics of PVS. From the induction hypothesis we then derive:

$$\text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(t'') = [\![trans(t'')]\!]\eta \quad .$$

If $[\![trans(t')]\!]\eta = \text{false}$, then $[\![trans(t)]\!]\eta = [\![trans(t''')]\!]\eta$ by the semantics of PVS. From the induction hypothesis we then derive:

$$\text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(t''') = [\![trans(t''')]\!]\eta \quad .$$

Note, that because $trans(t')$ is well-typed in PVS, we may conclude from the induction hypothesis: $\text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(t') \neq \mathcal{J}_{\mathbb{B}}$.

Therefore, in any case we conclude:

$$\text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(\textbf{if } t' \textbf{ then } t'' \textbf{ else } t''' \textbf{ endif}) = [\![\text{IF } trans(t') \text{ THEN } trans(t'') \text{ ELSE } trans(t''') \text{ ENDIF}]\!]\eta \quad .$$

6. Let $t, t_0, \ldots, t_n$ are OCL expressions, $v_0, \ldots, v_n$ variable names, and $\vec{v}_0, \ldots, \vec{v}_n$ lists of variable declarations (of the form $v : T$, where v is a variable name and T is a type), and $T_0, \ldots, T_n$ type names. By observing that the OCL evaluation function implements an environment-based, applicative order, and left-to-right evaluating abstract machine for a typed lambda, and the fact that **let** and LET have the same reduction semantics, we can conclude:

$$\begin{aligned}[\![trans(\textbf{let } v_0(\vec{v}_0) : T_0 = t_0, \ldots, v_n(\vec{v}_n) : T_n = t_n \textbf{ in } t)]\!]\eta &= \\ [\![\text{LET } v_0(\vec{v}_0) : T_0 = t_0, \ldots, v_n(\vec{v}_n) : T_n = t_n \text{ IN } t]\!]\eta &= \\ \text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(\textbf{let } v_0(\vec{v}_0) : T_0 = t_0, \ldots, v_n(\vec{v}_n) : T_n = t_n \textbf{ in } t) \quad .\end{aligned}$$

7. a) Let a be an attribute name, t be an OCL expression, and, as an induction hypothesis, we assume $\text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(t) = [\![trans(t)]\!]\eta$. By assumption, $trans(t.a)$ is well typed in PVS, which implies $[\![trans(t)]\!]\eta \neq null$. As a consequence $[\![trans(t.a)]\!]\eta$ is not undefined. Then we have by Definition 2.19:

$$\text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(t.a) = \xi(\text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(t))(a)$$

and by Definition 4.2 we have:

$$trans(t.a) = \text{state}(trans(t))\text{`aval}(\text{pvsattrname}(t, a)) \quad .$$

Using Lemma 4.5 we conclude:

$$\text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(t.a) = [\![trans(t.a)]\!]\eta \quad .$$

b) For navigation expressions $t.e$, where t evaluates to an object identity, assume as induction hypothesis, that $\text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(t) = [\![trans(t)]\!]\eta$. Then we have by Definition 2.19 and using Lemma 2.12:

$$\text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(t.e) = \iota x : (\text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(t)) \overset{e}{\rightarrow} x$$

and by Definition 4.2 we have:

$$trans(t.e) = \text{state}(trans(t))\text{`rval}(\text{pvsattrname}(t, e)) \quad .$$

Using Lemma 4.5 we conclude:

$$\text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(t.e) = [\![trans(t.e)]\!]\eta \quad .$$

8. For expressions of the form $t.v@\textbf{pre}$ the proof is similar to Case 7.

9. Let m be an operation name and $t_0, t_1, \ldots, t_n$ be a sequence of OCL expressions such that, as induction hypothesis, we have $\text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(t_i) = [\![trans(t_i)]\!]\eta$ for all $0 \leq i \leq n$. Furthermore, assume that $trans(t_0.m(t_1, \ldots, t_n)$ is well-typed in PVS. This implies, that $[\![trans(t_0)]\!]\eta \neq null$. Furthermore, because all type-checking constraints are provable, which implies that in PVS the expression $trans(m)(trans(t_0), trans(t_1), \ldots, trans(t_n))$ is not undefined for all interpretations of m. Consequently:

$$\text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(t_0.m(t_1, \ldots, t_n) \neq \mathcal{J}$$

for any undefined value $\mathcal{J}$.

Note, that proving $\text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(t_0.m(t_1, \ldots, t_n)) = [\![trans(t_0.m(t_1, \ldots, t_n))]\!]\eta$ has to be done for each operation defined in the model and its translation in PVS individually. The proof obligation is that the interpretation of m in OCL agrees with its translation in PVS *and* that the translation is type consistent if and only if the interpretation of m in OCL is not undefined. As an example, we show the case of integer division /.

The semantics of the operator / and its translation $trans(/) = /$ agree on real numbers. We have to prove that in $t_0./(t_1)$ it holds: $\text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(t_1) \neq 0$. Since $trans(t_0./(t_1)) = trans(t_0)/trans(t_1))$ is well typed in PVS, there exists a proof of $[\![trans(t_1)]\!]\eta \neq 0$. From the induction hypothesis $\text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(t_1) = [\![trans(t_1)]\!]\eta$ we infer $\text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(t_1) \neq 0$. Ergo, $\text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(t_0./(t_1)) = [\![trans(t_0/t_1)]\!]\eta$ holds.

10. The proof for the case of *collection property call expressions* is analogous to the case of *operation property call expressions* (Item 9).

11. If there are OCL expressions t_0, t, and t_n, types T and T_0, and variables v_0 and a such that $\varphi = t_0 \to \mathit{iterate}(v_0 : T_0; a : T = t \mid t_n)$ and

$$\begin{aligned}\mathit{trans}(t_0 \to \mathit{iterate}(v_0 : T_0; a : T = t \mid t_n)) = \\ \text{iterate}(\mathit{trans}(t_0), \mathit{trans}(t), \text{LAMBDA } (v_0 : \mathit{trans}(T_0), a : \mathit{trans}(T)) : \\ \mathit{trans}(t_n)) \quad ,\end{aligned}$$

then, using the assumption that the translation is well-typed, it holds, that

a) $\mathit{trans}(t_0)$ is well typed, and by definition of iterate, see Equation (4.1), a *finite* collection.

b) $\mathit{trans}(t)$ is well typed, therefore its value is in $\mathfrak{D}(\mathit{trans}(T)) = \mathfrak{D}(T)$.

c) LAMBDA $(v_0 : \mathit{trans}(T_0), a : \mathit{trans}(T)) : \mathit{trans}(t_n))$ is well typed, therefore its value is in $\mathfrak{D}(\mathit{trans}(T') \to \mathit{trans}(T) \to \mathit{trans}(T))$. As a consequence of the induction hypothesis $\lambda x.\lambda y.\, \text{eval}(D, \overleftarrow{\sigma}, \sigma, E\{v_0 \mapsto x, a \mapsto y)(\varphi') \in \mathfrak{D}(T' \to T \to T)$.

Consequently, $\mathit{trans}(\varphi) \in \mathfrak{D}(T)$.

12. Let t be an OCL expression such that $\text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(t) = [\![\mathit{trans}(t)]\!]\eta$ holds as induction hypothesis. Using an induction over the definition of *flatten*, see Item 12 of Definition 2.19, and flatten, see Definition 4.2, we conclude

$$\text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(t \to \mathit{flatten}()) = [\![\mathit{trans}(t \to \mathit{flatten}())]\!]\eta \quad .$$

13. For all OCL type expressions T the propositions

$$\text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(T.\mathit{allInstances}()) = [\![\mathit{trans}(T.\mathit{allInstances}())]\!]\eta$$

and

$$\text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(T.\mathit{allInstances}@\mathbf{pre}()) = [\![\mathit{trans}(T.\mathit{allInstances}@\mathbf{pre}())]\!]\eta$$

are a direct consequence of the Definition 2.19 and Lemma 4.5.

In any case, the claim holds. □

We now have the tools to prove the next lemma, which strengthens the property formulated in Lemma 4.6. The preceding lemma states, that if the translation of a constraint is well-typed in PVS, then the constraint has the same value as the translation in PVS. Next, we show that, if a constraint is well-typed in OCL and its value is not undefined, then it has the same value as its translation to PVS.

Corollary 4.7. *Let D be a class diagram and σ and $\overleftarrow{\sigma}$ be two object diagrams forming an action. Then all OCL expressions t such that* $\mathrm{eval}(D, \overleftarrow{\sigma}, \sigma, \epsilon)(t) \neq \mathcal{J}$ *holds satisfy* $\mathrm{eval}(D, \overleftarrow{\sigma}, \sigma, \epsilon)(t) = [\![\mathit{trans}(t)]\!]$.

Remark 4.1. It is important to observe that Corollary 4.7 does not state anything about the provability of the type correctness constraints in PVS. There is indeed no guarantee that all true type correctness constraint have a proof, because the logic of PVS is incomplete in standard models.

Next, we are going to establish the first soundness result:

Theorem 4.8. *Let t be an OCL constraint such that for all object diagrams $\overleftarrow{\sigma}, \sigma$ forming an action such that* $\mathrm{eval}(D, \overleftarrow{\sigma}, \sigma, \epsilon)(t) \neq \mathcal{J}$ *holds. Then* $\vdash \mathit{trans}(t)$ *implies* $\mathrm{eval}(D, \overleftarrow{\sigma}, \sigma, \epsilon)(t) = \mathrm{true}$.

Proof. Using Assumption 4.3 we conclude from $\vdash \mathit{trans}(t)$, that $\mathit{trans}(t)$ evaluates to true. Using Lemma 4.7 we conclude $\mathrm{eval}(D, \overleftarrow{\sigma}, \sigma, \epsilon)(t) = \mathrm{true}$. □

Observe that the assumption $\mathrm{eval}(D, \overleftarrow{\sigma}, \sigma, \epsilon)(\varphi) \neq \mathcal{J}$ is crucial in Theorem 4.8. This assumption mainly excludes constraints like the one of Example 4.3, whose value is undefined in OCL, but true in PVS.

Example 4.3. Consider the constraint $\mathit{Integer.allInstances}() \to \mathit{forAll}(\mathrm{true})$. We have that the following holds: $\mathit{trans}(\mathit{Integer.allInstances}() \to \mathit{forAll}(\mathrm{true})) = \forall(x : \mathrm{int}) : \textsc{true}$. This constraint is provable in PVS:

$$\frac{\vdash \forall(x : \mathrm{int}) : \textsc{true} \quad x' : \mathrm{int}}{\textsc{true}}\ \text{Skolem}$$

On the other hand, we have $\mathrm{eval}(D, \overleftarrow{\sigma}, \sigma, \epsilon)(\mathit{Integer.allInstances}() \to \mathit{forAll}(\mathrm{true}) = \mathcal{J}$ for all class diagrams D and all object diagrams $\overleftarrow{\sigma}, \sigma$, because the set of integers, described as $\mathit{Integer.allInstances}()$ in OCL, is an infinite set. ♦

The previous example demonstrates that the definition of quantifiers in OCL is finitary, while PVS does not have this limitation. In practise, the interpretation of quantifiers in PVS is preferable, because it allows to quantify over, for example, all integers.

Theorem 4.9. *Let φ be an OCL constraint such that* $\vdash \mathit{trans}(\varphi)$. *Then for all object diagrams $\overleftarrow{\sigma}, \sigma$ we have* $\mathrm{eval}(D, \overleftarrow{\sigma}, \sigma, \epsilon)(\varphi) \in \{\mathrm{true}, \mathcal{J}\}$.

Proof. Assume $\mathrm{eval}(D, \overleftarrow{\sigma}, \sigma, \epsilon)(\varphi) = \mathrm{false}$. Then $[\![\mathit{trans}(\varphi)]\!] = \mathrm{false}$ by Corollary 4.7. Assuming the soundness of PVS and the fact $\vdash \mathit{trans}(\varphi)$, we also have $[\![\mathit{trans}(\varphi)]\!] = \mathrm{true}$, which is a contradiction. Therefore $\mathrm{eval}(D, \overleftarrow{\sigma}, \sigma, \epsilon)(\varphi) \neq \mathrm{false}$. □

By Example 4.3 we already know of constraints which result in undefined in OCL. However, only the choice of translating *forAll* and *exists* to their respective quantifiers cause this problem.

4.7 Summary and Conclusion

The formalism and the methods described in this chapter have been implemented in a prototype compiler. We have tested it by, for example, verifying the object-oriented version of the *Sieve of Eratosthenes* described in Section 4.4. A trace-based proof is described in Chapter 6. Another proof using the compiler, which also establishes the liveness property that every prime will eventually be generated, has been established by Tamarah Arons using TL-PVS [129].

The complexity of the transition relation generated by our compiler proved to be challenging. It appears that this complexity is inherent to the UML semantics. The most difficult part is reasoning about messages in queues. The concept of messages preceding one another, crucial for the sieve, is difficult to work with for purely technical reasons. The proof of the sieve depends on the facts that no signals are ever discarded and that signals are taken from the queue in a first-in-first-out order. These two properties have to be specified in PVS as invariants and proved separately.[7] If one of the two properties does not hold, then the sieve does not satisfy its specification.

The run-time of the translator is usually less than a minute. The runtime is dominated by the time required to prove the model correct in PVS in any case. The coarse level of specifications in OCL and the expressive power of OCL is not sufficient to automate the whole verification process. For the proofs, annotations of the states of a state machine expressing invariants are highly useful, because these have to be formulated as intermediate steps in the current proof. This extension entails some changes of the semantic representation, as the proof method resulting from this is more similar to Floyd's inductive assertion networks (see [45] for references) as implemented in [43].

The work reported by Traore [148] defines a formalisation of UML state machines in PVS, which has been implemented in the PrUDE program. This program also implements a translation of state machines into PVS, but the semantics of state machines used by Traore differs from ours. This program also lacks the translation of OCL constraints to PVS.

The USE tool, described in, among others, the thesis of Mark Richters [133], implements an interpreter of OCL for run-time checking. USE only implements a way for testing OCL constraints, but not for verifying them using a theorem prover.

Brucker and Wolff describe a formalisation of OCL in Isabelle/HOL [101], another theorem prover for higher order logic [19, 20]. Contrary to our approach, they have extended partial functions to extended total functions by introducing an undefined value. This entailed the extension of the logic used in Isabelle/HOL to a three-valued logic, amounting to a deep embedding of OCL in the theorem prover. While this approach allows the verification of meta theorems, the verification of an actual UML model with respect to its OCL specification still requires the additional proof that all values do not correspond to undefined.

[7] This is not a limitation of PVS but a limitation of interactive theorem proving in general.

Chapter 5

Trace-based Compositional Specification and Verification for Object-Oriented Systems

The Object Constraint Language (OCL) is the established language for specifying properties of objects and object structures. OCL 2.0 has been extended with features that allow the specification of messages sent between objects.

We present a generalisation of this extension that allows to additionally specify causality constraints. From a pragmatic point of view, such causality constraints are needed to express, for example, that "each acknowledgement must be preceded by a matching request" in the context of exchanging messages, which is frequently required by communication protocols.

Our generalisation is based on the introduction of histories into OCL. Histories describe the external behaviour of objects. Moreover, to reason compositionally about the behaviour of a complex system we distinguish between *local* specifications within a single object and *global* specifications describing the interaction between objects. These two types of specifications are expressed in syntactically different dialects of OCL.

5.1 Introduction

OCL is used, among others, to constraining object structures in UML. Constraints on an object structure are invariants over a state of that structure. Such invariants use an object's attributes and the relationships between objects.

The behavioural concepts in OCL are pre- and postconditions of operations. The additional behavioural concept of a *message expression* has been introduced in OCL 2.0 [113]. Message expressions are predicates on traces of a particular object. As we shall demonstrate this imposes a real restriction.

We introduce a general framework for the behavioural specification of objects, a framework for modular specifications, and a compositional verification method for OCL. Behaviour is described in terms of messages and histories of events.

OCL 2.0 introduces a history as a sequence of local snapshots which are part of the interpretation of an object's valuation [113, pp. 5-4–5-5]. This history is not the kind of history we describe here. The history defined in OCL 2.0 is only part of the semantic domain of OCL and has no *syntactic* representation in OCL itself. It is therefore impossible to use this history in an OCL specification. In OCL 2.0 there is no type which is interpreted by Sequence(LocalSnapshot), and there is no expression whose value is the history of an object.

Adhering to the encapsulation principle of object-oriented programming we separate specifications into a *local* part describing the behaviour within an object, and a *global* part describing the message exchange between different objects to achieve a common goal. More precisely, a local specification consists, as its first part, of a specification of the internal structure of an object. The second part of a local specification, which we call *local interface specification* or *behavioural specification*, provides a specification of the observable behaviour of an object, its *local* history. A global specification specifies how the objects are associated to each other and how they *exchange* messages using a *global* history.

The compositional verification method introduced in this chapter is based on a compatibility predicate over local histories and the global history, which states that the composition of objects is feasible and the globally specified history is achieved. The compatibility predicate introduced in this chapter is a generalisation of the compatibility predicate for CSP described by Soundararajan in [144] for object-oriented systems. The verification step of checking the compatibility predicate relies only on the observable behaviour of objects and not on any specification of their internal structure. This enables the use of the compatibility test to identify design errors during early stages of the design.

The specification language used to specify a program's components may only use predicates over their *observable behaviour*. According to the encapsulation principle, a *behavioural* specification language should never state properties of the interior construction of the constituent components, for example, the underlying execution platform.

The encapsulation mechanisms of object-oriented design support the decomposition in a natural way. Object-oriented design makes the interface of the parts of a large system explicit; aggregation, a kind of association, is a way to group objects into larger parts. However, OCL does not enforce this encapsulation. To allow this we require the separation of specifications into local and global properties, as suggested by America [5] and Baumeister et al [9]:

1. A *local* specification is an OCL constraint on the local attributes and the local history of events of a single object, only.

2. A *local behavioural* specification is a local specification which only constrains the local history of an object and does not refer to the object's internal structure.

3. A *global* specification is an OCL constraint on the links, that is, references, *between* objects; a global specification constraints constrains the sequence of messages passed via these channels.

Local behavioural specifications and global specifications are used to separate concerns: The local behavioural specification is used to specify the behaviour of a program's constituents whereas the global specification is used to specify how these constituents interact. The local specification constrains the interior construction of an object. For a compositional specification we only need local behavioural specifications and global specifications.

The remainder of this chapter is structured as follows: In the next section we summarise the extension of OCL 2.0 with message expressions and demonstrate a weakness of this extension. In Section 5.3 we describe the notion of events and histories used in our extension of OCL. We identify the *observable behaviour* of an object for the assertional specification of that object. We use the traditional choice of messages sent and received by an object. In Section 5.4 we describe how our method facilitates compositional specifications and reasoning. We elaborate on the distinction between local specifications, local behavioural specifications, and global specifications. In Section 5.5 we describe our compositional verification method. Finally, in Section 5.6 we draw some conclusions and compare our results to related work.

5.2 State of the Art and Motivation

We describe the means for specifying interaction between objects in OCL 2.0, and explain in what respect they are not sufficient from a pragmatic point of view. To explain this we use the Example of the *Sieve of Eratosthenes*, which we have introduced in Section 4.4, and follow the ideas presented by America in [5].

Whether the property that the value of each attribute p of an instance of Sieve is a prime number is satisfied by the model depends on the behaviour of the other objects within the system. For example, we require that the generator does not send the value 1 to a sieve object, that the generator sends numbers in monotonically increasing order, and that the messages sent to a sieve object are *received* in monotonically order. This last requirement is the reason for our extension.

OCL 2.0 introduces the two operators ^ and ^^ for reasoning about messages. The first one, $o\hat{}msg(e)$, is a predicate that reads: "The contextual instance has sent a message msg to an object o with parameter values e." It is a predicate stating that a message has been sent during the execution of an operation. As such it should only be used in the postcondition of an operation specification. The predicate is true if such a message has been sent during the execution of *that* operation and false otherwise.

If the Sieve class of our example were to define an *operation e*, we could specify the behaviour of this operation as in the following example:

Example 5.1. The specification

context *Sieve* :: $e(z : Integer)$
pre : $z > p$
post : $itsSieve \rightarrow notEmpty()\ implies\ itsSieve\hat{}e(z)$

means that if the received value z is greater than p, and the contextual object is associated to a successor, then it will send an e message to the successor with the value z. ♦

Our example does not use any operation call, so we cannot say anything about the behaviour of Sieve using current OCL. The intended behaviour can, for example, be specified by a state machine (see Figure 4.3), but we want to specify the behaviour of the object on a higher level of abstraction. We also want to separate the obtained specification from the object's environment.

Here we need a notation stating that the contextual instance has *received* a message. To order the event of receiving and sending messages we also need *histories*.

To allow more complex reasoning about messages, OCL 2.0 introduces the message operator ^^. The expression $o\hat{}\hat{}msg(e)$ reads as "The sequence of all messages *msg* sent to an object o with parameter values e during the lifetime of the contextual object." This operator projects on the history of all messages sent by the contextual object to a specific object with specific parameter values. We can use this operator to, for example, require that the sequence of messages sent by an instance of Sieve is monotonically increasing.

Example 5.2. The expression

context *Sieve*
inv : **let** $message : Sequence(OclMessage) = \quad itsSieve\hat{}\hat{}e(? : Integer)$ **in**
$Integer\{2..(message \rightarrow size())\} \rightarrow forAll(i \mid$
$message \rightarrow at(i).z > message \rightarrow at(i-1).z)$

intends to state that the sequence of messages sent to the next sieve consists of strictly monotonically increasing values. ♦

But how do we specify that the sequence of messages *received* has to be monotonically increasing? There is no language construct in OCL 2.0 which allows one to assert that an object has received a message during the execution of an operation. Clearly, we cannot specify that, in order to send a message to another object, we need to have received a specific signal. There are no means in OCL to specify the messages which we have received.

OCL has introduced some interesting notions for specifying behaviour. But OCL 2.0 proposal allows only the specification of behaviour by referring to sent messages in an operation's postcondition. We envisage that practise requires more expressive means to specify more general properties of a system's behaviour, for example, causality constraints.

5.3 Observables

We introduce a more general formalism for specifying behaviour in OCL. We prefer to reason in terms of sequences of observable events, called histories. This makes it possible to specify that invoking an operation is always preceded by receiving a signal. In this section we define our notion of an observable event in UML and OCL.

Messages represent signals or operations by their names and by the actual arguments sent. Events correspond to sending or receiving a message. To keep the presentation simple, we only consider the events of sending and receiving asynchronous signals, and invoking and returning from an operation. This notion of observable events can be refined further to take the event queue of active objects into account, to model object creation and synchronous signals. In our setting, events correspond to the source and the target of arrows in sequence diagrams.

We restrict our presentation to reliable communication, that is, no message which is sent will get lost. Also, we assume an interleaving model for concurrency.

5.3.1 Events

Events drive the computation in UML models. An *event* is a specification of a kind of observation, for example, calling an operation or receiving a return value. The observation of an event is assumed to take place without duration at an instant in time. We distinguish two kinds of events: signal events and call events.

An *asynchronous signal* models the asynchronous communication between two objects. The associated events are sending the signal and receiving the signal.

A *synchronous signal*, formerly known as an *operation call*, models a synchronous communication between two objects. The associated events are sending (and at the same time receiving) the signal and *returning* from the reaction to the signal.

All kinds of events are represented using the data structure *OclEvent*.

Events are different from actions, which may cause sending or receiving of messages, and messages. An action may give rise to many events to be observed. The action of sending a signal allows one to observe that a signal is sent by one object or that the same signal is later received by another object.[1]

A message is a representation of a particular signal or call which is passed from one object to another. A message is part of many events. The same message may be observed both to be sent and to be received.

Communication Record.

In this section we explain the basic data structure which describes the observation of an event, called *communication record*. Communication records are instances of the class

[1] We avoid the complication of observing whether receiving a signal triggers a transition of a state machine, is discarded, or deferred.

OclEvent together with the *OclMessage* associated to the event, as shown in Figure 5.1.

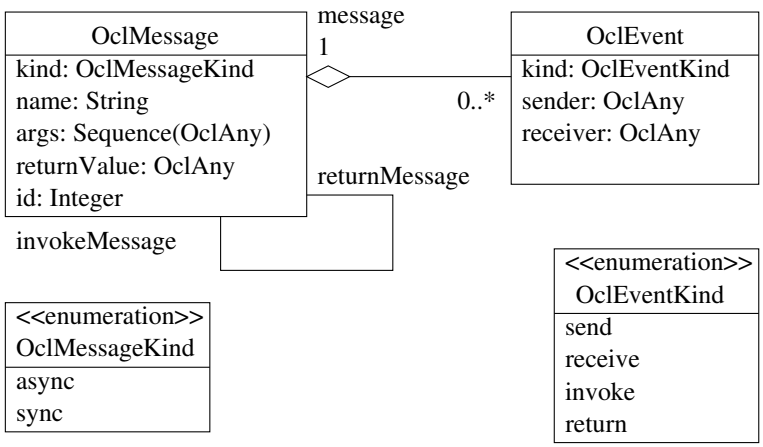


Figure 5.1: Definition of a communication record

An instance of *OclMessage* may be shared between many instances of *OclEvent*, because the same message may be part of many observations. On the other hand, each instance of *OclEvent* observes *exactly* one message. The enumeration OclMessageKind determines whether an instance of *OclMessage* refers to a message that has been sent asynchronously, indicated by the enumeration literal async, and thereby represents a signal, or whether it has been sent synchronously, indicated by the enumeration literal sync, and thereby represents an operation call. The following observations can be made when sending an asynchronous signal:

1. Sending a message representing the signal: This is represented by an instance of OCL event, where the attribute kind has the value of *send*.

2. Receiving a message representing the signal: This is represented by an instance of OCL event, where the attribute kind has the value of *receive*.

The case of synchronous messages, which represent operation calls, we observe two related events:

1. Invoking an operation by sending a message: This is represented by an instance of OCL event, where the attribute kind has the value of *invoke*.

2. Returning from an operation call by sending a return message: This is represented by an instance of OCL event, where the attribute kind has the value of *return*.

Send, receive, invoke, and return are literals of the enumeration *OclEventKind*. The multiplicity of 1 for the association from *OclEvent* to *OclMessage* models that each *OclEvent* refers to exactly one message.

A communication record, which is an instance of *OclEvent*, not only records the kind of the event, but also the information relating to its observation. It has the following attributes:

- Its *sender*.
- Its *receiver*.
- Its *message*: For an operation call two messages are sent: The first message is sent to initiate a call and consists of the operation's name and the actual parameters passed to the receiver. The second message is sent to indicate the completion of the call and to send return values back. For a signal only one message is sent.

  A message consists of:

  - Its *kind*, which is either *async* or *sync*.
  - Its *name*: The name of the operation or signal involved in this message.
  - Its *arguments*: A list of values representing the actual arguments sent with the message. A return message contains the actual parameter values of the invoking message along with the return value.
  - Its *return value*: This attribute holds the return value of an operation call.
  - Its *id*: This expresses a value used to uniquely identify a message. We assume a global counter which assigns to each message a unique integer in monotonically increasing order. This allows one, for example, to specify overtaking of messages.
  - Its *event*: This lists the events with which this message is associated. A message is generally associated with more than one event.
  - Its *invokeMessage*: If the message is a return message, the initiating invoke message can be accessed through this association.
  - Its *returnMessage*: If the message is a call message which has returned, then the return message can be accessed through this association.
- Its *kind*: One of the values from the *life cycle* associated with the message. If the message represents a signal, the value is either *send* or *receive*. If the message represents an operation call, the value is either *invoke* or *return*.

Before we proceed with our presentation, we introduce some notation. Whenever a value of the communication record is *undefined*, we write $\perp$ for this value. If an element e occurs in a sequence s, we write $e \in s$. If a sequence s is a *subsequence* of t we write $s \sqsubseteq t$.[2] We call an event *externally observable* if it can be observed

[2] A sequence s is a *subsequence* of t, if there exists a strictly monotonic function f such that $s_i = t_{f(i)}$ for all i.

from another object. This is the case whenever sender and receiver of a message are different. Messages sent by an object to itself are not observable.

We point out that our definition and the use of *OclMessage* is different from the *OclMessage* defined in OCL 2.0 [113]. We discuss the relation between these data types in Section 5.3.3.

Asynchronous Signal Events.

We record signal events with one communication record in the sender's history and one communication record in the receiver's history. The value of the attribute *kind* of the communication record may be:

OclEventKind::send This value models that a signal was sent from the sender to the receiver, and appears only in the sender's history.

OclEventKind::receive This value models that the signal has been received by the receiver, and appears only in the receiver's history.

Example 5.3. Assume two objects called g and s. If g sends a signal called e with the actual parameter values n to s, we have the following records in their histories:

$$\langle \mathit{OclEventKind} :: \text{send}, g, s, i, \langle \mathit{OclMessageKind} :: \text{async}, e, \langle n \rangle, \bot \rangle \rangle \in g.\mathit{localHistory}$$

and

$$\langle \mathit{OclEventKind} :: \text{receive}, g, s, j, \langle \mathit{OclMessageKind} :: \text{async}, e, \langle n \rangle, \bot \rangle \rangle \in s.\mathit{localHistory} \quad .$$

Note that i and j are integer variables identifying the event such that $i < j$. We have omitted the values for *invokeMessage* and *returnMessage* which are both undefined. ♦

Operation Calls.

An operation call causes two synchronisations between the sender and receiver: First it synchronises when the operation is invoked, and then it synchronises when the operation completes and a return value is sent back. This is modelled by two *synchronous* messages: the first message is used to invoke the operation, the second is used to send the return value of the call back and to report completion of the operation call. The communication records appear in both the sender's and receiver's history, because they refer to synchronous communications . The value of the attribute kind of the communication record is:

OclEventKind::invoke This value models that a synchronous signal was sent by the sender and received by the receiver in order to invoke an operation.

OclEventKind::return This value models that a synchronous signal was sent by the sender and received by the receiver in order to indicate the completion of the operation call and to send back the return value.

Example 5.4. Assume the existence of two objects called o and p. If o calls an operation named m with the actual parameter values $\vec{a}$ of the object p, which then returns the value v, we have the following subsequences of their histories:

$$\langle\langle \mathit{OclEventKind} :: \text{invoke}, o, p, i_0, \langle \mathit{OclMessageKind} :: \text{sync}, m, \vec{a}, \bot\rangle\rangle,$$
$$\langle \mathit{OclEventKind} :: \text{return}, p, o, i_3, \langle \mathit{OclMessageKind} :: \text{sync}, m, \vec{a}, v\rangle\rangle\rangle \sqsubseteq$$
$$o.\mathit{localHistory}$$

and

$$\langle\langle \mathit{OclEventKind} :: \text{invoke}, o, p, i_1, \langle \mathit{OclMessageKind} :: \text{sync}, m, \vec{a}, \bot\rangle\rangle,$$
$$\langle \mathit{OclEventKind} :: \text{return}, p, o, i_2, \langle \mathit{OclMessageKind} :: \text{sync}, m, \vec{a}, v\rangle\rangle\rangle \sqsubseteq$$
$$p.\mathit{localHistory} \quad .$$

For any j the i_j express integers identifying the event such that for their values we have $i_j < i_{j+1}$. ♦

5.3.2 History

We have defined the events relating to sending and receiving of signals and calling operations. Sequences of such events are called *histories*. Histories are instances of *Sequence*(*OclEvent*). Therefore, we can reuse the operations defined for *Sequence* in the standard library (see Section 2.3.4). These operations are sufficient for almost all uses.

We distinguish between a *local* and a *global* history. A local history contains the externally observable events of a *single* object and describes its interface. A global history contains all observable events of the complete system. These observable events are all events generated by all objects of a system during its computation and the events received from its environment, in the order in which they occur.

The local history is introduced into OCL by extending the class *OclAny* with the attribute *localHistory* of type *Sequence*(*OclEvent*). Recall that *OclAny* is the (implicit) base class of all other classes, see Section 2.3.4, which implies that *all* objects inherit the attribute *localHistory*. A global history of the system is supplied in a similar manner using the attribute *globalHistory*. This is shown in Figure 5.2.

5.3.3 Comparison to OCL 2.0

We argue that our notion of history in OCL is at least as expressive as the message expressions proposed in OCL 2.0. First we show how the data types used for mes-

context *OclAny*
inv : *localHistory* → *forAll*(*e* | (*e*. sender = *self or e*. receiver = *self*) *and*
(*e*. sender <> *e*. receiver)
inv : *globalHistory* → *forAll*(*e* | *e*. sender <> *e*. receiver)
inv : *OclAny.allInstances*() → *forAll*(*o*, *p* | *o.globalHistory* = *p.globalHistory*)

Figure 5.2: Properties of histories

context *OclMessage* :: hasReturned() : *Boolean*
post : *result* = (kind = return *or* returnMessage → *size*() > 0)

context *OclMessage* :: result() : *T*
pre : kind = return
post : *result* = returnValue

context *OclMessage* :: isSignalSent() : *Boolean*
post : *result* = (kind = sent)

context *OclMessage* :: isOperationCall() : *Boolean*
post : *result* = (kind = invoke *or* kind = return)

T refers to the return type of the operation the message refers to.

Figure 5.3: Properties of OCL 2.0's *OclMessage*

sage expressions can be represented in our formalism. Then we explain how OCL 2.0 message expressions can be expressed as history expressions. We shall point out some semantic ambiguities in the standard, and how they may be corrected.

OclEvent and OCL 2.0's OclMessage

OCL 2.0 defines its own data type *OclMessage* [113, p. 6-4], which is syntactically different from ours. Here we explain their relation. The four operations on *OclMessage* defined in OCL 2.0 can be specified as in Figure 5.3.

The ^ Operator

As described in Example 5.1 and in Section 2.7.1 on page 2-23 of [113], a message expression using the ^ operator results in an instance of Boolean. It evaluates to true if and only if a message m with arguments $\vec{a}$ has been sent during the execution of its contextual operation. To this end, we define the history of events sent and received dur-

ing the execution of an operation, that is, the sequence of all events between receiving an invoke and the corresponding return event:

let last : *OclEvent* = *localHistory* → *at*(*localHistory* → *size*()) **in**
let start : *Integer* =
  let search(*i* : *Integer*) : *Integer* =
    if *i* ≥ 0 *and not*
      (*localHistory* → *at*(*i*).*message.name* =
        localHistory → *at*(last).*message.name*
      and localHistory → *at*(*i*).*message*.args =
        localHistory → *at*(last).*message*.args
      and localHistory → *at*(*i*).kind = *OclEventKind* :: *invoke*
      and localHistory → *at*(*i*).sender = *localHistory* → *at*(last).receiver)
    then search(*i* − 1) **else** *i* **endif**
  in search(*localHistory* → *size*())

In a postcondition of an operation last refers to the index of the event sending the return message back to the caller in the callee's local history, whereas start refers to the index of the corresponding invoke message of this operation call. Then *localHistory* → *subSequence*(start, last) expresses the sequence of all events observed during the execution of the contextual operation. Since we need only to check that the send event with the message *m* and with arguments $\vec{a}$ occurs in this subsequence, the meaning of *o*^*msg*(*e*) is:

localHistory → *subSequence*(start, *localHistory* → *size*()) →
  select(*m* | *m.state* = *send and m.receiver* = *o and*
    m.message.name = 'msg' *and m.message.args* = *e*) → *notEmpty*()

By this we have established:

Proposition 5.1. *The OCL 2.0 expression o^msg(e) is expressible as a local history expression.*

The ^^ Operator

The exact semantics of the ^^ operator is not precisely defined in the OCL standard; we base this discussion on the description of this operator on page 2-23 of the standard proposal [113] and in the paper by Kleppe and Warmer [78]. Recall, that the informal meaning of this operator is: "The sequence of all messages of a specific name sent to a specific object with some specific parameter values during the lifetime of the contextual object." The standard proposal provides the following example as an explanation of the ^^ operator:

> **context** Subject :: hasChanged() **post** : observer^^update(12, 14). This results in the Sequence of messages sent. Each element of the collection is an instance of *OclMessage*.

It is not clear whether this expression refers to the sequence of messages sent during the life time of the contextual instance or only to the sequence of messages sent during the execution of the contextual operation. Because it is used in the same way as the ^ operator, we assume the latter.

The semantics of the expression in our formalism is (analogous to Section 5.3.3):

$$\begin{aligned} &\mathit{localHistory} \rightarrow \mathit{subSequence}(\text{start}, \mathit{localHistory} \rightarrow \mathit{size}()) \rightarrow \\ &\quad \mathit{select}(m \mid m.\mathit{state} = \mathit{send}\ \mathit{and}\ m.\mathit{receiver} = o\ \mathit{and} \\ &\qquad m.\mathit{message}.\mathit{name} = \text{`msg'}\ \mathit{and}\ m.\mathit{message}.\mathit{args} = e).\mathit{message} \end{aligned}$$

Note the application of the navigation operator to message at the end of the constraint. This expression is not a sequence of instances of *OclEvent* but a sequence of instances of *OclMessage*, which (according to Section 5.3.3) is obtained as in the proposed standard. We note this result as:

Proposition 5.2. *The OCL 2.0 operator o^^msg(e) can be expressed by a history expression.*

Proposition 5.1 and Proposition 5.2 imply that OCL 2.0 message expressions can also be expressed by history expressions.

Corollary 5.3. *History expressions are at least as expressive as OCL 2.0 message expressions.*

5.4 Local and Global Specifications

We have shown that, in addition to our data types and a mechanism which updates histories, everything needed for behavioural specifications is already present in standard OCL. In this section we show that our proposed extension is more general than the message expressions of OCL 2.0.

For our specification language we need quantification over sets and sequences, quantification over the set of all subsequences of a sequence, and projection operations applied to sequences.

- Quantification over collections are provided by the usual *forAll* and *exists* operations.
- Quantification over subsequences of a sequence can be expressed by using a function that computes the set of all subsequences of a sequence.

- Projection operations are expressed using *select* and *collect* operations.

We refine the concept of histories in OCL by the notion of local and global specifications. A *local* specification is a constraint on the internal structure of a *single* object. A *local behavioural* specification is a constraint on the externally observable behaviour of a single object, expressed as a constraint on its local history. A *global* specification is a constraint on the links, which describe whether one object can send a message to another object, between a group of objects, and the values communicated along these links. This distinction is introduced to separate different concerns:

- Local specifications are used to specify the behaviour of a single object in any possible environment in which this object can be used. Such specifications are used to constrain the instance variables of an object. Most important is that such a specification is not part of the interface of an object, because the constrained attributes are not part of the interface. A client of an object need not have any knowledge of these constraints.

- Local behavioural specifications are constraints on the local history of an object.

  Observe, that local specifications are *not* constraints on local histories.

- Global specifications are used to specify the relations between different objects, that is, they constrain the values of associations, and how different objects collaborate to achieve a common goal. In the global specification we constrain the environment of an object.

This separation of concerns is one of the fundamental contributions of component-based design. We consider each object also as a component. When considered as such, the signals it is able to receive and the operations it defines are the interface which this component *provides*. By specifying which messages an object *sends*, the interface it *requires* is made explicit, because it describes a requirement on the messages the *receiver* has to provide.

5.4.1 Local Specification Language

A local specification is a specification which refers only to the attributes of an object, the values of its associations, and its local history. Expressions which contain arbitrary navigation operators are not allowed as local specifications. It is only allowed to test association ends of an objects for equality or containment, whether such an association end is empty, or how many elements can be reached through it.

For our Sieve example, local specifications are

context *Generator*
inv : $x > 1$

context *Sieve*
inv : *not oclInState*(s_0) *implies* $p > 1$
inv : *not oclInState*(s_0) *implies Integer*$\{2..(p-1)\} \rightarrow$ *forAll*($i \mid p \bmod i \neq 0$)

Quantification is bounded by local collections. A local collection is only constructed from an OCL expression which do not contain navigation and *allInstances* expressions; for example, *Integer*$\{2..(p-1)\}$ expresses a local collection. We may construct such a collection from the local history, local attribute values, and the local association relations:

context *Sieve*
inv : *self* $\neq$ *itsSieve*
inv : *oclInState*(s_1) *implies itsSieve* $\rightarrow$ *notEmpty*()

This restriction is imposed, because we want to make sure that the local specification refers only to the locally known object identities and the identities the object may have known in the past. The following is *not* a local constraint:

context *Sieve*
inv : *oclInState*(s_1) *implies itsSieve.p* $> p$
inv : *oclInState*(s_1) *implies* (*itsSieve.oclInState*(s_1)
implies itsSieve.itsSieve $\rightarrow$ *notEmpty*())

The first invariant asserts that the object known to it as *itsSieve* has a value for p which is greater than its own. This is a statement about the other object, too.

Local specifications usually constrain the implementation of an object. As such, the client of such an object should not need to know about implementation details. For example, it is not necessary to know that the behaviour of an object is implemented or specified by a state machine.

To describe the interface of an object we introduce local behavioural specifications. These are specifications that only uses the local history of an object.

The reason for this restriction is that any client of an object may only invoke operations or send signals specified in the object's interface. Then the client can at most observe the messages the object sends. The client cannot observe the state of an object, if this is not specified in the interface. Because the purpose of a local behavioural specification is to only specify its observable behaviour there is no need to refer to the encapsulated state. Also, documenting and specifying the non-observable part of an object in its interface can be confusing for programmers, who want to use the object as a ready-made component. For example, a behaviour of an instance of Sieve can be

described by:

context *Sieve*
inv : *Integer*{1..*localHistory* → *size*()} → *forAll*(*i* |
localHistory → *at*(*i*).*kind* = *send and*
localHistory → *at*(*i*).*message.name* = ‘e’
implies Integer{1..*i*} → *exists*(*j* | *localHistory* → *at*(*j*).*kind* = *receive and*
localHistory → *at*(*j*).*message.name* = ‘e’ *and*
localHistory → *at*(*j*).*message.arguments* → *at*(1) =
localHistory → *at*(*i*).*message.arguments* → *at*(1))))

This means that an instance of Sieve only sends an *e* signal with a value if it has received one before it. This is a (weak) causality constraint.[3] The observable behaviour serves as an abstraction of the internal state. Given a history of events of an object we can simulate the object's behaviour and therefore determine its internal state, once we know its initial state. All information necessary to do this is represented in this local history.

In our previous example, we do not refer to the sender of the signal received or the receiver of the signal sent. This cannot be done with state machines or message diagrams. It is, on the other hand, useful during top-down design, because we can postpone the decision of to whom we send the signal, or even decide to send it to self.

Another constraint on the history of our sieve example is, that each sieve object receives messages in increasing order. This can be expressed as:

context *Sieve*
inv : **let** recv = *localHistory* → *select*(*e* | *e.receive and e.name* = ‘e’) **in**
Integer{2..recv → *size*()} → *forAll*(*i* | *localHistory* → *at*(*i* − 1).*message.args*
→ *at*(1) < *localHistory* → *at*(*i*).*message.args* → *at*(1))

5.4.2 Global Specification Language

Objects do not function alone. They usually interact with other objects. We specify how an object collaborates with its environment using a global specification language. This level of specification has two purposes:

- We can define global constraints on how the objects collaborate.
- We can give further constraints on how objects are linked together in a certain application. These global constraints are called component invariants by Baumeister et al in [9].

[3]The constraint is weak, because it allows that one send observation may cause an arbitrary number of receive observations. The constraint can be strengthened to require that for each receive we have exactly one receive. See Section 6.5.2 for examples of how this may be specified.

A global specification usually should not constrain an object's attributes, but it should constrain the links between objects, similar to architecture diagrams.

Example 5.5. Recall the sieve example. Then a global specification stating that the generator is associated to the sieve object with the smallest number is:

$$\textbf{context } \mathit{Generator} \textbf{ inv} : \mathit{Sieve} \to \mathit{allInstances}() \to \mathit{forAll}(s \mid \mathit{itsSieve}.p \leq s.p)$$

⧫

Global specifications may refer to the global history. For instance, stating formally that a message will be passed on to a different object is a global specification.

Example 5.6. Recall the Sieve example. One global property of the system is that if a number is not a prime it will eventually not be retransmitted.

$$\begin{array}{l}
\textbf{context } \mathit{Generator} \\
\textbf{inv} : \mathit{Integer}.\mathit{allInstances}() \to \mathit{forAll}(n \mid n > 1 \textit{ implies} \\
\quad \mathit{Sieve}.\mathit{allInstances}() \to \mathit{exists}(s \mid \mathit{globalHistory} \to \mathit{forAll}(e \mid \\
\qquad (e.\mathit{sender} = s \textit{ and} \\
\qquad\quad e.\mathit{kind} = \mathit{OclEventKind} :: \text{send } \textit{and } e.\mathit{message}.\mathit{args} \to \mathit{at}(1) = n) \\
\qquad \textit{implies } \mathit{Integer}.\mathit{allInstances}() \to \mathit{exists}(m \mid \mathit{globalHistory} \to \mathit{at}(m) = e \textit{ and} \\
\qquad\quad \mathit{Integer}.\mathit{allInstances}() \to \mathit{forAll}(\ell \mid \ell > m \textit{ implies } \mathit{globalHistory} \to \\
\qquad\qquad \mathit{at}(\ell).\mathit{message}.\mathit{args} \to \mathit{at}(1) \neq n)))))
\end{array}$$

Unfortunately, OCL does not allow suitable abbreviations in this formula. It reads: "For every integer n there exists an instance s of Sieve such that for each message e with argument n sent by s there exists a position m at which e occurs in the global history and for any position l after m in the global history the value n does not occur as an argument." ⧫

5.5 Compatibility

Our purpose is to verify the feasibility of the composition of objects by proving consistency of a set of local behavioural specifications through the use of a *compatibility predicate*. Compatibility essentially means that a set of objects has a global computation. This means that for its local histories we can find a matching global history.

The main benefit of using a compatibility predicate is that it enables us to verify correctness properties of the system under development during the early stages of its design. For compatibility we only need:

1. A specification of the globally desired behaviour as specified by an invariant on the global history,
2. a specification of how the objects of the system are composed and how they communicate, as specified by a global invariant, and

3. the externally observable behaviour of each object in the system as specified by an invariant on their local histories.

We do not need any further knowledge about the implementation of the objects in the system (for example, about how those objects are composed themselves) and we can leave the specification of the implementation of these objects for later stages of the development process.

Essentially, the compatibility predicate establishes whether an n-tuple of local histories $(\chi_1, \ldots, \chi_n)$ of the participating objects $o_1, \ldots, o_n$ fit together in the sense that there exists a global history of their composition which, when projected on the composing object o_i, yields the original local history χ_i of the object one started out with.

Definition 5.4. Let O be a set of objects. To each $o \in O$ we assign a local history χ_o. We write $\vec{\chi}$ for the list of local histories χ_o for $o \in O$ and $\text{proj}_o(\chi)$ for the projection of the history χ on the object o. Then the compatibility predicate is defined by:

$$\text{compat}(\vec{\chi}) \stackrel{\text{def}}{=} \exists \chi . \bigwedge_{o \in O} \text{proj}_o(\chi) = \chi_o \quad . \tag{5.1}$$

◊

A UML model, however, does not specify the system in terms of objects but in terms of classes. We reformulate the compatibility predicate in terms of classes.

Given a class diagram consisting of classes $C_1, \ldots, C_n$ with associated local behavioural specifications $\varphi_1, \ldots, \varphi_n$, and given a global invariant Φ on the object structure and the global history, we extend the definition of the compatibility predicate (5.1) as follows:

$$\exists \chi . \Phi \wedge \bigwedge_{i=1}^{n} \forall z_i . \varphi_i[z_i/self] \wedge z_i.localHistory = \text{proj}_{z_i}(\chi) \tag{5.2}$$

In this expression, the variable χ denotes a global history, the expression $\varphi_i[z_i/self]$ denotes the result of substituting each occurrence of *self* by z_i, and z_i is an instance of C_i. This Definition (5.2) lifts a local property of an object to the global level.[4]

The predicate described in Equation (5.2) can be expressed in OCL except for the existence of the global history χ. We use the name *globalHistory* as a Skolem-constant for it. The following expression describes the compatibility predicate in OCL:

context *OclAny*
inv : globalHistory → *select*(*e* | *e*. sender = *self or e*. receiver = *self*) = localHistory

[4]For technical convenience we assume that all attributes used in the local behavioural specifications are explicitly qualified with self. Otherwise, the substitution operation used later has to explicitly handle the use of unqualified attributes.

The lifting of the local constraints to the global level is implicitly done in the semantics of OCL provided by Richters [133]. The global invariant can be formulated as an invariant of any class or "spread" among the classes.

To use the compatibility test it is sufficient to provide a constraint for the local history (if one is missing, we assume true, which is any history) and add the above constraint to *OclAny*. Besides writing suitable constraints on local histories, the compatibility test is easily implemented. The tool described in Chapter 4 has been extended to handle these trace specifications which enables the *verification* of the compatibility test using PVS.

5.6 Conclusions, Related Work, and Future Work

The message expressions discussed in this paper are introduced in OCL 2.0 [113]. Kleppe and Warmer define the semantics of the action clause in terms of histories of events [78]. This paper inspired our definition of histories. We have shown that the proposal for message expressions in OCL fails to take received messages into account. We have introduced an extension of OCL which does take these received messages into account. Our formalism, based on histories of events, can emulate the message expressions of OCL 2.0 and is more expressive.

The idea of taking those messages which an object receives into account during specifying a system is commonly accepted in component-based design. Successful applications of this way of modelling are presented by Selic et al in [142], and have been integrated into UML 2.0 [111].

An early extension of OCL with means to reason about received and sent events is presented in the Catalysis Approach by D'Souza and Wills [49]. D'Souza and Wills add some syntactic constructs which allow the modeller to specify that messages have been sent in OCL. Their extension lacks the possibility to state that two messages have been sent in a particular order, because they do not define a history. Bradfield, Küster Filipe, and Stevens introduce histories of events into OCL and specifications are written in the observational μ-calculus [17]. This extension of OCL allows one to reason about messages sent and received. But the authors do not allow the specifier to use the full power of their extension. The authors advocate a use of specification patterns. Experience with Bandera shows, that a library of patterns may become larger than the language on top of which they are build on.

Very similar to our extension is the one by Cengarle and Knapp [22]. They introduce histories of time-stamped events and modal operators like *always* and *eventually*. Our formalism is lacking means to specify liveness properties and only permits to specify of safety properties.

A very similar formalisation of OCL message expressions has been independently developed by Flake and Mueller [54]. While they define the semantics of OCL message expressions in terms of sequences of observations, similar to our local histories, they

do not introduce the concept of a history into the OCL. Consequently, they are not able to reason about received messages. However, they indicate that there is a need for temporal extensions of OCL.

Since we extend our formalism to compositional specification of real-time systems in Chapter 7, the only liveness property to require for real-time systems is progress of time (see Hooman [64] for references). All the extensions of OCL to sequences of events known to us do not consider compositional specifications, for which the specification language has to be *adapted* appropriately.

The verification method described in this paper is based on the adaptation of the definition of the compatibility predicate for communicating sequential processes [144].

Our extension of OCL is conservative. It does not add any new syntactic constructor to the language and does not lead to any change of the OCL meta-model. Instead, histories and events are introduced as new data types and their operators can be defined within existing OCL. Using this approach makes the meaning of a message expression immediately apparent to users of OCL. The drawback is that such expressions are often hard to read.

For certain settings it is even possible that compatibility can be checked without interaction of a user. One way to achieve this is the use of a tableaux method [143]. Tableaux methods allow the construction of counter-examples to a specification, as in model-checking techniques [28, 130].

Chapter 6

A Compositional Trace Logic for Behavioural Interface Specifications

We describe a compositional trace logic for behavioural interface specifications and corresponding proof rules for compositional reasoning. The trace logic is defined in terms of axioms in higher-order logic. This trace logic is applicable to any object-oriented programming language. We treat object creation without observing the explicit act of creation.

We prove a soundness result of this approach using the theory of Galois connections. We show the correctness of a specification of the Sieve of Eratosthenes using the proposed method. This notion of compositionality allows the verification of systems during the early stages of a design.

6.1 Introduction

We present a theory for reasoning compositionally about behavioural interfaces for class-based object-oriented programs. Our main contribution is an axiomatic characterisation of unbounded object creation in terms of communication traces over the visible operations of a class (its *signature*). This involves an abstraction of the actual creation of objects as represented explicitly in, for example, Ábrahám [3] for multi-threaded Java.

We apply this method to verifying the correctness of the Sieve of Eratosthenes prime number generator using PVS (cf. Owre [121]). This example involves unbounded object creation.

The next section characterises our notion of an *interface* and *interface invariants*. Section 6.3 describes the *trace logic* used to specify interface invariants. Section 6.4 explains how to reason compositionally about object-oriented systems using interface invariants. Section 6.5 describes the axioms of our trace logic and contains a correctness result based on Galois connections. Section 6.6 applies our method to the "Sieve of Eratosthenes" example. Section 6.7 relates our work to previously published work. Finally, in Section 6.8 we draw our conclusion on our method and discuss future work.

6.2 Interfaces

An *interface* defines the interaction between software components by means of properties of other software components, which abstract and encapsulate their data. This includes constants, data types, and the signature of synchronous and asynchronous message exchange, including exceptions specifications. The interface of a component is deliberately kept separate from its implementation. Any other software component B, which can be referred to as the client of A, that interacts with A is forced to do so only through the interface. The advantage of this arrangement is that any other implementation of the interface of A should not cause B to fail, provided that its use of A complies with the specification of A's interface. Synchronous operations are executed by means of the standard rendez-vous mechanism, resulting in synchronous message exchange; asynchronous operation calls are executed by storing a corresponding message in the receiver's message queue.

Furthermore, an interface specifies a *protocol* between objects, that is, how unrelated objects communicate with each other. In our case a protocol is a description of:

- The messages understood by an object.
- The arguments that these messages may be supplied with.
- The results that these messages return, if any.
- The *invariants* that are preserved despite of the modifications to an object's state.
- The exceptional situations that will be required to be handled by clients of the object.
- The allowed sequence of messages.
- Restrictions placed on either participant in the communication.
- Expected effects that will occur if a message is handled.

In this paper, interfaces are specified using *interface invariants*. These are invariants on *traces of events*, that is, sequences of occurrences describing message sending and receiving, using the trace logic of the following section. Note that whereas we require that interface specifications are compatible with the class of each object, we do not discuss type-checking of interface specifications. Therefore, we do not define the signature of a message, an operation call, or a class.

6.3 Trace Logic

Trace logic is used to specify and verify properties of the externally observable behaviour of objects. A trace denotes a sequence of events which indicate sending and receiving of messages. It provides a state-less abstraction of the behaviour of an object.

The assertion language for describing properties of traces is an order-sorted logic with equality. It includes the elementary data types of the underlying programming language, for example, integers and booleans.

The names of operations specified in interfaces are only used in the definition of the additional abstract data type of *events*. Each event e has the following attributes: $e.\,\mathrm{name}$ denotes the name of the operation of the event, $e.\,\mathrm{sender}$ its sender, $e.\,\mathrm{receiver}$ its receiver, $e.\,\mathrm{args}$ its sequence of arguments, and $e.\,\mathrm{kind}$ indicates

- sending an asynchronous message (by *send*),
- reading an asynchronous message from the message queue (by *recv*),
- sending and receiving an operation call (by *call*), or
- sending and receiving a return of an operation call (by *return*).

The abstract data type of a trace is modelled by a sequence of events. We assume a distinguished variable θ of this type, which denotes a global trace of a system of objects.

The semantics of this language is standard, except for the interpretation of variables ranging over objects, whose domain are the objects occurring in θ.

Observe that in the proposed data type event we do not have a kind that describes the creation of objects. It is often not necessary to reason about object creation. Instead, one reasons about the interactions between objects. These interactions are constrained by the *object-structure* of the system, that is, the objects' knowledge of other objects. This knowledge may either be constrained using an invariant or inferred from the local traces by observing the use of other objects. Consequently, we abstract from object creation. In Section 6.5.1, however, we demonstrate that under certain circumstances it is necessary to observe object creation. In most cases the implicit knowledge of object creation contains complete information about the object structure and the acquaintance relation between objects.

We use the following notations:

The empty sequence is expressed by ϵ.

For $n \in \mathbb{N}$ define $\mathrm{below}(n) \stackrel{\mathrm{def}}{=} \{m \in \mathbb{N} \mid m < n\}$.

s_i refers to the $(i+1)$th element of the sequence s, whose length is expressed by $|s|$.

$s \preceq s'$ states that s is a prefix of s'.

For any sequence s and any natural number $n \leq |s|$ define $\mathrm{prefix}(s, n)$ to be the prefix of s with length n, that is, $\mathrm{prefix}(s, n) \preceq s$ and $|\,\mathrm{prefix}(s, n)| = n$.

Analogously, we define $\mathrm{suffix}(s, n)$ to be the suffix of s starting at position n.

$s \downarrow S$ expresses the *projection* on S, that is, the largest subsequence of s over elements from S. We also write $s \downarrow e$ instead of $s \downarrow \{e\}$.

A trace θ is called *local to an object o*, if o is the sender of any send event and the receiver of any read event, that is,

$$\mathrm{local}(\theta, o) \stackrel{\mathrm{def}}{=} \theta \downarrow \{e \mid \varphi(e)\} \quad ,$$

where $\varphi(e)$ is:

$$(e.\,\text{kind} = \mathit{send} \implies e.\,\text{sender} = o) \wedge (e.\,\text{kind} = \mathit{recv} \implies e.\,\text{receiver} = o) \quad .$$

For later use we list here some abstractions of a global trace θ, which are defined by suitable projection operations:

$$\begin{aligned}
\text{recvby}(\theta, o) &\stackrel{\text{def}}{=} \theta \downarrow \{e \mid e.\,\text{kind} = \mathit{recv} \wedge e.\,\text{receiver} = o\} \\
\text{recvfrom}(\theta, o) &\stackrel{\text{def}}{=} \theta \downarrow \{e \mid e.\,\text{kind} = \mathit{recv} \wedge e.\,\text{sender} = o\} \\
\text{sentby}(\theta, o) &\stackrel{\text{def}}{=} \theta \downarrow \{e \mid e.\,\text{kind} = \mathit{send} \wedge e.\,\text{sender} = o\} \\
\text{sentto}(\theta, o) &\stackrel{\text{def}}{=} \theta \downarrow \{e \mid e.\,\text{kind} = \mathit{send} \wedge e.\,\text{receiver} = o\}
\end{aligned}$$

The local assertion language is used for specifying local properties of the behaviour of an object and is obtained from the global assertion language by introducing a special logical variable *self*, denoting the object under consideration, and *interpreting* occurrences of θ as referring to $\text{local}(\theta, \mathit{self})$.

Additionally, we do not allow unbounded quantification over objects, but require that quantification over objects is bounded by the objects occurring in the trace.

Given a logical variable o, the substitution $[o/\mathit{self}]$ transforms a local assertion into a global one by replacing every occurrence of *self* by o and every occurrence of θ by $\text{local}(\theta, o)$.

6.4 Compositionality

After having introduced our trace logic, we describe our verification method. It is based on introducing local assertions $\mathcal{I}_c$ as interface invariants for each class $c \in C$, where C is the set of all classes occurring in the system, whereas the global assertion language is used to specify global properties ϕ of the communication network. Then the proof obligation for proving ϕ is expressed by the following formula, explained in a number of steps:

$$\mathfrak{A} \vdash \bigwedge_{c \in C} \forall z_c : \mathcal{I}[z_c/\mathit{self}] \implies \phi(\theta), \tag{6.1}$$

where $\mathfrak{A}$ axiomatises the theory of global traces, as described in Section 6.5, and where:

1. z_c expresses a variable ranging over instances of class c and the quantification, as described in the previous section, ranges over all instances of class c occurring in global trace θ, and

2. the substitution $[z_c/\mathit{self}]$ replaces every occurrence of *self* by z_c.

Observe that the substitution $[z_c/self]$ enforces in (6.1) that the local trace $\theta \downarrow self$ of the object denoted by z_c equals the projection of $\text{local}(\theta, z_c)$. Note that this substitution expresses compatibility between the local interface invariants in terms of the global trace θ.

Using this proof obligation we derive from the local specifications of each class a global property. This makes the proof method compositional.

6.5 Axiomatisation

In this section we describe the axioms of our trace logic, which can be used to prove properties of a system.

6.5.1 Observing Object Creation

In our trace logic we do *not* observe the creation of objects as such, because we claim that the creation of an object can be inferred from the global trace.

Intuitively, we infer that object o has created o' if it *reveals* o'. An object o' is revealed by o in a trace θ when o sends in θ a message to o' or it sends a message with o' as one of the parameter values without having received o' as a parameter of a communication record in θ before.[1]

$$\begin{aligned}\text{reveal}(\theta, o, o') \stackrel{\text{def}}{=} \exists i : \text{kind}(\theta_i) \neq \mathit{recv} \\ \wedge\ \text{sender}(\theta_i) = o \wedge (\text{receiver}(\theta_i) = o' \vee o' \in \theta_i.\,\text{args}) \\ \wedge\ \forall j < i : \theta_j.\,\text{receiver} = o \implies o' \notin \theta_j.\,\text{args} \quad (6.2)\end{aligned}$$

Define $\text{child}(\theta, o)$ as the set of children of o, where we call o' a *child* of o if o reveals o' in θ:

$$\text{child}(\theta, o) \stackrel{\text{def}}{=} \{o' \mid o \neq o' \wedge \text{reveal}(\theta, o, o')\}$$

The first axiom expresses that an object cannot have been revealed by two different objects in a trace θ.

Axiom 6.1. $\forall o, o' : o \neq o' \implies \text{child}(\theta, o) \cap \text{child}(\theta, o') = \emptyset$.

The second axiom of our trace logic prevents cycles in the chain of creation. To formalise this, we define the transitive closure of the child relation and call it offspring:

$$\text{offspring}(\theta, o) \stackrel{\text{def}}{=} \text{child}(\theta, o) \cup \{o' \mid \exists o'' \in \text{child}(\theta, o) : o' \in \text{offspring}(\theta, o'')\}$$

Axiom 6.2. $\forall o : o \notin \text{offspring}(o, \theta)$.

[1] We write $v \in \vec{v}$ if there is an index i such that $\vec{v}_i = v$.

Note that we do not require that every object has been created by another object. Such objects are called *root objects*. This implies that in a trace there can appear more than one root object. Such traces represent computations starting from a initial configuration containing possibly more than one root object.

Soundness

In this section we establish the soundness of our approach and derive precise conditions under which our abstraction is sound. We prove the soundness of our approach by establishing a Galois connection between traces with object creation and objects without object creation. We define the notion of Galois connection.

Definition 6.3. Let $(P, \leq)$ and $(Q, \sqsubseteq)$ be ordered sets. A pair (α, γ) of maps $\alpha : P \to Q$ and $\gamma : Q \to P$ is called a *Galois connection* between P and Q if for all $p \in P$ and $q \in Q$:

$$\alpha(p) \sqsubseteq q \Leftrightarrow p \leq \gamma(q) \tag{6.3}$$

◊

Following this definition we have to define two partially ordered sets of traces. Our trace logic is modelled by the set of traces without object creation. We order this set discretely, that is, the set of traces without object creation is ordered by equality.

Next, we introduce traces with object creation.

Definition 6.4. Let create(o, o') represent the observation that o creates o'. A trace θ is a *trace with object creation*, if every object occurring in a trace is created at most once and whenever o communicates with o', then o has received the identity of o' before or o has created o', that is, the following axioms hold:

$$\forall i, j : \exists o, o' : (\theta_i = \text{create}(o, o')) \wedge (\theta_j = \text{create}(o, o')) \implies (i = j) \tag{6.4}$$

$$\begin{aligned} \forall o : \forall i : (\theta_i.\,\text{kind} = \mathit{send} \vee \theta_i.\mathit{kind} = \mathit{call}) \implies \\ \forall o' \in \{\theta_i.\,\text{receiver}\} \cup \theta_i.\,\text{args} : o = o' \vee \\ \Big(\exists j < i : (\theta_j = \text{create}(o, o')) \vee \\ ((\theta_j.\,\text{kind} = \mathit{recv} \vee \theta_j.\,\text{kind} = \mathit{call}) \wedge \\ o = \theta_j.\,\text{receiver} \wedge o' \in \theta_j.\,\text{args})\Big) \end{aligned} \tag{6.5}$$

and

$$\forall o' : \exists i, o : \theta_i = \text{create}(o, o') \implies \forall j < i : o' \notin \theta_j \quad , \tag{6.6}$$

where $o' \notin \theta_j$ states that o' does not occur in any of the fields of θ_j. Additionally, we require that traces satisfy Axiom 6.2, that is, the relation of object creation, is acyclic. ◊

Equation (6.4) corresponds to Axiom 6.1, namely, that each object is created by at most one other object.

Equation (6.5) asserts that whenever an object reveals another object, then it has created this object before.

Finally, Equation (6.6) states that each object that is created by another object is actually created before its first activity.

We observe the following property:

Remark 6.1. For any trace θ with object creation, if $\text{reveal}(\theta, o, o')$ holds then there is an index i such that $\theta_i = \text{create}(o, o')$ and $\neg\, \text{reveal}(\text{prefix}(\theta, i), o, o')$.

Equation (6.5) is stating the same fact as (6.2) states.

For convenience, we write

$$\text{nocreate}(\theta) \stackrel{\text{def}}{=} \theta \downarrow \{e \mid e.\,\text{kind} \neq \text{create}\}$$

and

$$\text{created}(\theta) = \{o \mid \exists i : \exists o' : \theta_i = \text{create}(o', o)\} \quad .$$

Now we define a partial order on traces with object creation. The intention is to consider a trace "larger" (more abstract) if it is "lazier" in creating new objects (that is, it creates objects later).

In the next definition we use the fact that projection can be expressed using a largest strictly monotonically increasing function f. We use the notation f^- for the inverse of a function.

Definition 6.5. Let θ and θ' be traces with object creation. We say that θ creates objects *more lazily* than θ', written $\theta' \leq \theta$, if and only if

$$\text{nocreate}(\theta) = \text{nocreate}(\theta') \tag{6.7}$$

and if f is the projection function from θ to $\text{nocreate}(\theta)$, g the projection function from θ' to $\text{nocreate}(\theta')$, then for all $i \in \text{below}(|\,\text{nocreate}(\theta)|)$

$$\text{created}(\text{prefix}(\theta, f^-(i))) \subseteq \text{created}(\text{prefix}(\theta', g^-(i))) \tag{6.8}$$

and

$$\text{created}(\theta) \subseteq \text{created}(\theta') \quad . \tag{6.9}$$

◊

Equation (6.7) states that we only order traces which represent the same communication behaviour.

Equation (6.8) allows the eager trace θ' to create more objects and to create them earlier. By Equation (6.5) these additionally created objects in θ' are redundant, because no messages are sent to them and their values are never communicated.

Equation (6.8) does not consider the case that the traces θ and θ' have suffixes which only consist of object creations, because the projection functions have proper communications in domain range and these suffixes do not occur in the range of the projection functions. Therefore, Equation (6.9) is required to assert that θ creates less objects or creates objects later than θ' in this suffix.

One important property of Definition 6.5 is that traces in which objects are created consecutively, but in a different order, are equivalent. In Equations (6.8) and (6.9) we compare the objects in the prefixes which represent the same communication behaviour.

When reconstructing a trace with object creation and we encounter a send event where more than one object is revealed, we have to guess an order, in which these objects have been created (as done in Equation (6.10)).

Because the order is not determined, we find traces θ and θ' with $\theta \neq \theta'$, $\theta \leq \theta'$ and $\theta' \leq \theta$. Consequently, we need a different notion of equivalence. Therefore, we write $\theta \gtrless \theta'$ if $\theta \leq \theta'$ and $\theta' \leq \theta$. One can easily establish the following remark, exploiting the fact that $\theta \leq \theta'$ and $\theta' \leq \theta$ expresses that θ and θ' differ in the order of consecutive create events:

Remark 6.2. $\gtrless$ is an equivalence relation.

Next, we establish that $\leq$ as defined in Definition 6.4 is indeed a partial order:

Lemma 6.6. *Let $\leq$ as defined in Definition 6.5. Then $\leq$ is a partial order on traces with object-creation modulo $\gtrless$.*

Proof. Apparently, $\leq$ is reflexive.

We prove that $\leq$ is transitive: Let $\theta \leq \theta'$ and $\theta' \leq \theta''$. From (6.7) we conclude $\text{nocreate}(\theta) = \text{nocreate}(\theta')$ and $\text{nocreate}(\theta') = \text{nocreate}(\theta'')$. Consequently, $\text{nocreate}(\theta) = \text{nocreate}(\theta'')$.

Let $i \in \text{below}(|\text{nocreate}(\theta)|)$, f the function projecting θ onto $\text{nocreate}(\theta)$, g the function projecting θ' onto $\text{nocreate}(\theta')$, and h the function projecting θ' onto $\text{nocreate}(\theta')$. Then Equation (6.8) implies $\text{created}(\text{prefix}(\theta, f^{-}(i))) \subseteq \text{created}(\text{prefix}(\theta'', h^{-}(i)))$.

From the assumptions we conclude $\theta \leq \theta' \implies \text{created}(\theta') \subseteq \text{created}(\theta)$ and $\theta' \leq \theta'' \implies \text{created}(\theta'') \subseteq \text{created}(\theta')$. The transitivity of $\leq$ implies $\text{created}(\theta'') \subseteq \text{created}(\theta)$.

Finally, we prove antisymmetry modulo permutations of consecutive create events. Let θ, θ' such that $\theta \leq \theta'$ and $\theta' \leq \theta$. From (6.7) we conclude that θ and θ' represent the same behaviour. Now we show that each event, except consecutive create events, occur

on the same position. Let $i \in \text{below}(|\,\text{nocreate}(\theta)|)$, f the function projecting θ onto $\text{nocreate}(\theta)$, and g the function projecting θ' onto $\text{nocreate}(\theta')$. Again, we conclude

$$\text{created}(\text{prefix}(\theta, f^{-}(i))) = \text{created}(\text{prefix}(\theta', g^{-}(i)))$$

from (6.8). It follows that θ and θ' create the same objects between the same events which are not a create event. Only a different order of these creates are allowed.

Analogous reasoning establishes Equation (6.9). □

As required by Definition 6.3 we have now defined two partially ordered sets, namely the set of traces without object creation, which we order discretely, and the set of traces with object creations ordered by $\leq$. Next we have to define two functions α and γ to establish the Galois connections. As α we use the function nocreate. Gamma is defined next. We need the following notations:

$$\text{start}(\theta) \stackrel{\text{def}}{=} \theta_0 \cdots \theta_{|\theta|-2}$$

and

$$\text{last}(\theta) \stackrel{\text{def}}{=} \theta_{|\theta|-1} \quad .$$

The operator $\cdot$ expresses concatenation of finite sequences. If no ambiguity arises, we treat elements as sequences of length 1 for the purpose of concatenation.

As γ we define a function that creates objects as late as possible. Before we define this function, we need some auxiliary definitions. Let θ be a trace with object creation, o and o' objects, and $i = \max\{j \mid o' \notin \theta_j\}$ if o' occurs in θ and $i = |\theta|$ if o' does not occur in θ. Then the function insert inserts a create-event into the trace before the *first* occurrence of o', or appends the event if o' does not occur in θ:

$$\text{insert}(\theta, o, o') \stackrel{\text{def}}{=} \text{prefix}(\theta, i) \cdot \text{create}(o, o') \cdot \text{suffix}(\theta, i)$$

Observe that it is possible to create any number of objects during one observation of a trace without object creation, as they all can be revealed by passing them as parameters. Therefore, we extend insert to inserting a set O of objects created by o as follows:

$$\text{insert}(\theta, o, O) \stackrel{\text{def}}{=} \begin{cases} \theta & \text{if } O = \emptyset \\ \text{insert}(\theta, o, o') & \text{if } O = \{o'\} \\ \text{insert}(\text{insert}(\theta, o, O \setminus \{o'\}), o, o') & \text{if } |O| > 1 \text{ and } o' \in O. \end{cases}$$

This extended "function" insert is actually not a function, because the order in which the elements of O are inserted is not determined. But the order in which the create-events are inserted is not relevant and these different orders are considered equal by $\gtrless$, as shown by the following Lemma:

Lemma 6.7. *Let θ express a trace with object creation, o some object and O a set of objects. Then all results of* $\mathrm{insert}(\theta, o, O)$ *are in an $\gtrless$-equivalence class.*

We define γ inductively as follows:

$$\gamma(\theta) \stackrel{\text{def}}{=} \begin{cases} \epsilon & \text{if } \theta = \epsilon \\ \gamma(\mathrm{start}(\theta)) \cdot \mathrm{last}(\theta) & \text{if } \mathrm{child}(\theta, o) = \mathrm{child}(\mathrm{start}(\theta), o) \\ \mathrm{insert}(\gamma(\mathrm{start}(\theta)), o, O) \cdot \mathrm{last}(\theta) & \text{otherwise,} \end{cases} \tag{6.10}$$

where $o = \mathrm{last}(\theta).\,\mathrm{sender}$ and $O = \mathrm{child}(\theta, o) \setminus \mathrm{child}(\mathrm{start}(\theta))$.

Before we prove the main result of this section, we first have to establish that the result of γ is indeed a valid trace with object creation.

Lemma 6.8. *Let θ be a trace without object-creation. Then $\gamma(\theta)$ is a trace with object creation, that is, it satisfies Definition 6.4.*

Proof. By induction on θ. Let θ be a trace without object creation.

Case $\theta = \epsilon$. Then $\gamma(\theta) = \epsilon$ and ϵ satisfies Definition 6.4.

Case $\theta \neq \epsilon$. Assume as an induction hypothesis $\gamma(\mathrm{start}(\theta))$ satisfies Definition 6.4. Let, for the remainder of the proof, $o = \mathrm{last}(\theta).\,\mathrm{sender}$. We distinguish two sub-cases:

If $\mathrm{child}(\theta, o) = \mathrm{child}(\mathrm{start}(\theta), o)$, then the last communication did not create any new objects. By induction hypothesis, we know that $\gamma(\mathrm{start}(\theta))$ satisfies Definition 6.4. Because $\mathrm{child}(\theta, o) = \mathrm{child}(\mathrm{start}(\theta), o)$ also $\gamma(\mathrm{start}(\theta)) \cdot \mathrm{last}(\theta)$ satisfies Definition 6.4.

If $\mathrm{child}(\theta, o) \neq \mathrm{child}(\mathrm{start}(\theta), o)$ holds, then $\mathrm{child}(\mathrm{start}(\theta), o) \subset \mathrm{child}(\theta), o)$ holds, too. Then $O = \mathrm{child}(\theta, o) \setminus \mathrm{child}(\mathrm{start}(\theta), o)$. Observe that O is not empty and finite. Next, we prove that $\mathrm{insert}(\gamma(\mathrm{start}(\theta)), o, O)$ satisfies Definition 6.4 by induction on O.

If $O = \emptyset$ then $\mathrm{insert}(\gamma(\mathrm{start}(\theta)), o, \emptyset) = \gamma(\mathrm{start}(\theta))$, which, by the first induction hypothesis, satisfies Definition 6.4.

Assume $O \neq \emptyset$. Let $o' \in O$, let $\theta' = \mathrm{insert}(\gamma(\mathrm{start}(\theta)), o, O \setminus \{o'\})$, and assume as induction hypothesis that θ' satisfies Definition 6.4. To prove: $\mathrm{insert}(\theta', o, o')$ satisfies Definition 6.4. From the assumption that o' is an object revealed by o and the creation of o' has not been inserted into θ' we conclude there is no o'' such that $\mathrm{create}(o'', o')$ occurs in θ'.

Let i be the largest number such that o' does not occur in $\mathrm{prefix}(\theta', i)$. If θ'_i exists, then θ'_i is the first occurrence of o', otherwise it is $\mathrm{last}(\theta)$. Therefore, in the trace $\mathrm{prefix}(\theta', i) \cdot \mathrm{create}(o, o') \cdot \mathrm{suffix}(\theta', i)$ only contains one $\mathrm{create}(o, o')$, as required by Equation (6.4) and since o' does not occur in $\mathrm{prefix}(\theta', i)$, $\mathrm{prefix}(\theta', i) \cdot \mathrm{create}(o, o') \cdot \mathrm{suffix}(\theta', i)$ also satisfies Equation (6.5). Consequently, $\mathrm{insert}(\theta', o, o')$ satisfies Definition 6.4. □

This lemma establishes the type-correctness of the function γ. Moreover, it demonstrates the necessary conditions under which one may abstract from object creation in a trace-based theory. The function γ is defined in terms of a function insert which inserts the necessary create events into a trace. The function does not insert the create

observation before the object which has to be created is revealed. The reason for this is that the created object is *active*, that is, it has its own thread of control and starts communicating with other objects as soon as it has been created.

In our theory passing parameters to an objects constructor method is modelled as ordinary communication. In this case, the newly created object waits for its creator to call the constructor method. But it need not wait but can, after initialising into a default state, create its own objects and communicate with them. In this case, an object may be revealed *after* it has sent messages. Within the prefix of a trace before the object has been revealed, it appears to be a root-object.

Consider a *set* of traces without object creation Θ characterising all computations of a system. Furthermore, assume that Θ is prefix-closed, as required by the theory of Zwiers [158]. Then the set of traces with object creation $\{\gamma(\theta) \mid \theta \in \Theta\}$ is generally not prefix-closed, because in the prefixes of traces in which an active object has been created, that object appears to be a root-object. For systems with active objects specified in our trace logic where specifications have to be prefix-closed, one either has to make sure that object-creation does not matter or one has to observe it, because it *cannot* be reconstructed using the function γ.

These considerations also imply that if we *require* that specifications are prefix-closed, these results could not have been established. For active objects one has often to know the continuation of a trace in order to decide whether an object has been created by another one.

Lemma 6.9. *For all traces θ without object creation* $\mathrm{nocreate}(\gamma(\theta)) = \theta$ *holds.*

Now we prove the main theorem of this section:

Theorem 6.10. *Let $\alpha =$* nocreate *and γ as defined in* (6.10)*. Then (α, γ) is a Galois connection between the set of traces without object creation and the set of traces with object creation.*

Proof. Case $\alpha(\theta) = \hat{\theta} \implies \theta \leq \gamma(\hat{\theta})$: Let θ be a trace with object creation and $\hat{\theta}$ a trace without object creation such that $\alpha(\theta) = \hat{\theta}$. Using Lemma 6.9 we obtain $\alpha(\theta) = \mathrm{nocreate}(\gamma(\hat{\theta}))$. It remains to prove (6.8). From the assumption $\hat{\theta} = \alpha(\theta)$ we find a projection function $f : \mathrm{dom}(\theta) \longrightarrow \mathrm{dom}(\hat{\theta})$. Let g be the projecting function from $\gamma(\hat{\theta})$ to $\hat{\theta}$. Let $i \in \mathrm{below}(|\hat{\theta}|)$. Assume there exists a $p \in \mathrm{created}(\mathrm{prefix}(\gamma(\hat{\theta}), g^-(i)))$ such that $p \notin \mathrm{created}(\mathrm{prefix}(\theta, f^-(i)))$. Then we find a smallest i' such that $p \in \mathrm{created}(\mathrm{prefix}(\gamma(\hat{\theta}), g^-(i')))$ and, as a consequence of (6.10), $\hat{\theta}_{i'}$ is the first event where p is revealed. Because $\hat{\theta} = \alpha(\theta) = \mathrm{nocreate}(\theta)$ we know that $\alpha(\theta)_{i'}$ is also the first event where p is revealed. However, we assumed $p \notin \mathrm{created}(\mathrm{prefix}(\theta, f^-(i)))$, which contradicts Definition 6.4. Therefore, there is no such p, and (6.8) holds. Equation (6.9) follows from the fact that $\gamma(\hat{\theta})$ never ends in a create event.

Case $\theta \leq \gamma(\hat{\theta}) \implies \alpha(\theta) = \hat{\theta}$: Let θ be a trace with object creation and $\hat{\theta}$ a trace without object creation such that $\theta \leq \gamma(\hat{\theta})$. Therefore $\alpha(\theta) = \alpha(\gamma(\hat{\theta}))$ by (6.7). Now

observe that $\alpha(\theta) = \text{nocreate}(\theta)$, from which we conclude $\alpha(\theta) = \alpha(\gamma(\hat{\theta}))$. Using Lemma 6.9 we finish the proof. □

The existence of a Galois connection between traces with object creation and without object creation asserts that properties specified on traces without object creation also hold for traces with object creation obtained by applying the function γ and traces which are more eager, that is, where object creation is observed earlier, than the traces obtained from γ. This also means that one can, if memory is not limited, abstract from object creation in specifications. In general, creating an object is not part of the behaviour. Similar ideas were presented by Cousot and Cousot in [36] and by Dams in [40] in the analysis of programs. The idea of lazy object creation has also been applied by Ábrahám et al in, for example, [2] for establishing a fully abstract semantics of object-oriented programs.

6.5.2 Communication Mechanisms

We assume that all asynchronous messages are received in the event queue in a first-in-first-out order. This is expressed by Axiom 6.11.

Axiom 6.11. $\forall o : \zeta(\text{recvby}(\theta, o)) \preceq \zeta(\text{sentto}(\theta, o))$, where ζ is the function which removes kind from all communication records.

Note that any other kind of asynchronous communication could be adopted as well.

Finally, observe that synchronous communication is modelled by the definition of local traces as projections of the global trace (cf. [158]).

Axiom 6.12. $\forall o : \zeta(\text{recvby}(\theta, o)) = \zeta(\text{sentto}(\theta, o))$, where ζ is the function which removes kind from all communication records.

We also assume that synchronous communication is non-reentrant, meaning that, after an object has accepted a call, it first has to return before accepting another call or receiving another message. This is expressed by the next axiom:[2]

Axiom 6.13.

$$\begin{aligned}\forall o : \forall i : \theta_i.\,\text{kind} = \mathit{return} \wedge \theta_i.\,\text{sender} = o \implies \\ \exists j < i : \theta_j.\,\text{kind} = \mathit{call} \wedge \theta_j.\,\text{receiver} = o \wedge \\ \theta_j.\,\text{sender} = \theta_i.\,\text{receiver} \wedge \theta_j.\,\text{name} = \theta_i.\,\text{name} \wedge \\ \exists v : \theta_j.\,\text{args} = \theta_i.\,\text{args} \cdot v \wedge \\ \forall k : j < k \wedge k \leq i : (\theta_k.\,\text{kind} = \mathit{call} \vee \theta_k.\,\text{kind} = \mathit{recv}) \implies \theta_k.\,\text{receiver} \neq o\end{aligned}$$

[2] This axiom expresses a communication mechanism used by UML state machines, as defined by the Object Management Group [111]. To obtain reentrant operation calls, this axiom has to be adapted.

6.6 Sieve of Eratosthenes

In this section we specify the "Sieve of Eratosthenes" and prove its correctness. The Sieve of Eratosthenes is an efficient algorithm for finding all prime numbers. The idea is to have a generator which sends the natural numbers starting with 2 to a sieve object. A sieve object decides, depending on the first number it has received, whether it may be a prime or not. If the number received is divisible by the first number it has received, then it is a composite number and cannot be a prime number. If the sieve object determines that the received number is not divisible by the first number received, it sends the number to the next sieve object.

More precisely, we have two classes: *Generator* and *Sieve*. Instances g of class *Generator* create exactly one instance of class *Sieve*, see (6.11) below, and then send it the increasing sequence $2, 3, 4, \ldots$ of natural numbers as specified by (6.12) below, identifying functions over $\mathbb{N}$ with sequences. Instances s of class Sieve also create exactly one instance of class Sieve by (6.11) and then "sift" the sequence of numbers they receive by (6.13), where this sifting process is recursively defined in (6.14).

$$\forall o : \exists o' : \mathrm{child}(\theta, o) \subseteq \{o'\} \tag{6.11}$$

$$\mathrm{sentby}(\theta, g).\,\mathrm{args} \preceq \lambda n.n + 2 \tag{6.12}$$

$$\mathrm{sentby}(\theta, o).\,\mathrm{args} \preceq \mathit{Sieve}(\mathrm{recvby}(\theta, o).\,\mathrm{args}, \mathrm{recvby}(\theta, o)_0.\,\mathrm{args}) \tag{6.13}$$

For s a sequence over $\mathbb{N}$ and $n \in \mathbb{N}$ define

$$\mathrm{sift}(s, n) \stackrel{\mathrm{def}}{=} \begin{cases} \epsilon & \text{if } s = \epsilon \\ \mathrm{sift}(\mathrm{tail}(s), n) & \text{if } n \mid \mathrm{head}(s) \\ \mathrm{head}(s) \cdot \mathrm{sift}(\mathrm{tail}(s), n) & \text{if } n \nmid \mathrm{head}(s), \end{cases} \tag{6.14}$$

where $n \mid s$ states that s divides n and $n \nmid s$ states that s does not divide n.

Here the class invariant $\mathcal{I}_{\mathit{Generator}}$ of *Generator* is the conjunction of (6.11) and (6.12) and $\mathcal{I}_{\mathit{Sieve}}$ is the conjunction of (6.11) and (6.13).

For each instance g of Generator we find a sequence of sieve instances which each have received a prime number as their first value. This pipe structure p denotes a sequence of objects determined by the global trace θ. It is inductively defined by the predicate

$$\Pi(\theta, p) \stackrel{\mathrm{def}}{=} \begin{cases} \mathrm{true} & \text{if } |p| \leq 1, \\ \mathrm{head}(\mathrm{tail}(p)) \in \mathrm{child}(\theta, \mathrm{head}(p)) \wedge \Pi(\theta, \mathrm{tail}(p)) & \text{otherwise,} \end{cases}$$

expressing that every object in the pipe only sends messages to its child, being its successor. If an object sends a message to objects other than its successor, the object has either received another object identity, which contradicts (6.12) and (6.13), because only the sending of numbers is allowed, or it has created a second child, which contradicts (6.11). Observe that the first object p_0 in p denotes the generator.

As required in (6.1) we now have the local invariants $\mathcal{I}_{Generator}$ (the conjunction of (6.11) and (6.12)) and $\mathcal{I}_{Sieve}$ (the conjunction of (6.11) and (6.13)), as well as the the global sifting property $\phi(\theta)$ defined in (6.15) below. Validity of the resulting formula is formulated in Theorem 6.14 and proved later. Observe that $\Pi(\theta, p)$ is not an additional assumption. As explained above, its validity can be derived from (6.11–6.13).

Theorem 6.14. *For any global trace θ satisfying (6.11–6.13) with p ranging over sequences of objects satisfying $\Pi(\theta, p)$ one has*

$$\forall i \in \text{below}(|p| - 1) : \text{sentby}(\theta, p_{i+1}).\,\text{args} \leq \text{sift}(\text{sentby}(\theta, p_i).\,\text{args}, \text{sentby}(\theta, p_i)_0.\,\text{args}) \quad (6.15)$$

Lemma 6.15. *For any sequences θ and θ' with $\theta \preceq \theta'$ and any n we have* $\text{sift}(\theta, n) \preceq \text{sift}(\theta', n)$.

Proof. By induction on θ and θ' (cf. also [81]). □

Next define

$$O(\theta) \stackrel{\text{def}}{=} \{o \mid \exists i : \text{sender}(\theta_i) = o \vee \text{receiver}(\theta_i) = o \vee o \in \theta_i.\,\text{args}\} \quad .$$

Lemma 6.16. *Let θ be a trace of the sieve system and p a sequence of objects such that $\Pi(\theta, p)$ holds. Then* $\text{sentto}(\theta, p_{i+1}) = \text{sentby}(\theta, p_i)$ *for all* $i \in \text{below}(|p| - 1)$.

Proof. By induction on the length of the trace θ.

If $|\theta| = 0$ then $\text{sentto}(\theta, p_{i+1}) = \epsilon$ and $\text{sentby}(\theta, p_i) = \epsilon$.

Suppose $|\theta| > 0$. With the assumptions $\Pi(\text{start}(\theta), p)$ and $\text{sentto}(\text{start}(\theta), p_{i+1}) = \text{sentby}(\text{start}(\theta), p_i)$ we distinguish two sub-cases:

Case $p_{i+1} \in O(\text{start}(\theta))$: The induction hypothesis states $p_{i+1} \in \text{child}(\text{start}(\theta), p_i)$.

Because $\text{last}(\theta).\,\text{kind} = \textit{recv}$ and because sentto and sentby observe only *send* events and no *recv* events, these equalities hold:

$$\text{sentto}(\theta, p_{i+1}) = \text{sentto}(\text{start}(\theta), p_{i+1})$$
$$\text{sentby}(\text{start}(\theta), p_i) = \text{sentby}(\theta, p_i)$$

Again, we distinguish two sub-cases:

Case $\text{last}(\theta).\,\text{kind} = \textit{send}$ and $\text{last}(\theta).\,\text{sender} \neq p_i$. From the fact $\text{child}(\theta, p_i) = \{p_{i+1}\}$, $\Pi(\text{start}(\theta), p)$, and from

$$\text{sentto}(\theta, p_{i+1}) = \text{sentto}(\text{start}(\theta), p_{i+1})$$

and, therefore,

$$\text{sentby}(\text{start}(\theta), p_i) = \text{sentby}(\theta, p_i) \quad ,$$

we conclude $\text{last}(\theta).\,\text{receiver} \neq p_{i+1}$.

If $\text{last}(\theta).\,\text{kind} = \mathit{send}$ and $\text{last}(\theta).\,\text{sender} = p_i$, then $\text{child}(\theta, p_i) = \{p_{i+1}\}$ (again, because $\Pi(\text{start}(\theta), p)$ holds) and therefore

$$\text{sentto}(\theta, p_{i+1}) = \text{sentto}(\text{start}(\theta), p_{i+1}) \cdot \mathit{send}(p_i, p_{i+1}, e, v)$$

and

$$\text{sentby}(\text{start}(\theta), p_i) \cdot \mathit{send}(p_i, p_{i+1}, e, v) = \text{sentby}(\theta, p_i) \quad ,$$

which proves the claim for this case.

Case $p_{i+1} \notin \mathcal{O}(\text{start}(\theta))$: The induction hypothesis states $p_{i+1} \notin \text{child}(\text{start}(\theta), p_i)$. Because of $p_{i+1} \in \text{child}(\theta, p_i)$, the last observation in θ is sending a message from p_i to p_{i+1}, that is, $\text{last}(\theta) = \mathit{send}(p_i, p_{i+1}, e, v)$ for some value v. Using the induction hypothesis and the definitions of sentto and sentby we have

$$\begin{aligned}\text{sentto}(\theta, p_{i+1}) = \text{sentto}(\text{start}(\theta), p_{i+1}) \cdot \mathit{send}(p_i, p_{i+1}, e, v) = \\ \text{sentby}(\text{start}(\theta), p_i) \cdot \mathit{send}(p_i, p_{i+1}, e, v) = \text{sentby}(\theta, p_i)\end{aligned}$$

In any case the claim holds. □

Proof. [Proof of Theorem 6.14] By induction on p.

Let θ be a trace and p a sequence of objects such that $\Pi(\theta, p)$ holds.

- If $|p| \leq 1$, that is, if only p_0 exists, the claim is trivially true.
- Assume $|p| > 1$, $p_1 \in \text{child}(\theta, p_0)$, and, as an induction hypothesis, that Theorem 6.14 holds for $\text{tail}(p)$.

  To prove:

  $$\text{sentby}(\theta, p_1).\,\text{args} \preceq \text{sift}(\text{sentby}(\theta, p_0).\,\text{args}, \text{sentby}(\theta, p_0)_0.\,\text{args}).$$

  - If $\text{sentby}(\theta, p_1) = \epsilon$, then the claim is trivially true.
  - Assume that $\text{sentby}(\theta, p_1) \neq \epsilon$. Then $\text{sentby}(\theta, p_1)_0$ and $\text{recvby}(\theta, p_1)_0$ are defined. Lemma 6.16 implies $\text{sentto}(\theta, p_1) = \text{sentby}(\theta, p_0)$ and Axiom 6.11 implies

    $$\text{recvby}(\theta, p_1) \preceq \text{sentto}(\theta, p_1),$$

    resulting in

    $$\text{recvby}(\theta, p_1) \preceq \text{sentby}(\theta, p_0).$$

Use Lemma 6.15, Axiom 6.11, and $\text{recvby}(\theta, p_1)(0) = \text{sentby}(\theta, p_0)(0)$ to obtain

$$\text{sift}(\text{recvby}(\theta, p_1).\,\text{args}, \text{recvby}(\theta, p_1)_0.\,\text{args}) \preceq \text{sift}(\text{sentby}(\theta, p_0).\,\text{args}, \text{sentby}(\theta, p_0)_0.\,\text{args})$$

Use the transitivity of $\preceq$ and Formula (6.13) as

$$\text{sentby}(\theta, p_1).\,\text{args} \preceq \text{sift}(\text{recvby}(\theta, p_1).\,\text{args}, \text{recvby}(\theta, p_1)_0.\,\text{args})$$

to obtain

$$\text{sentby}(\theta, p_1).\,\text{args} \preceq \text{sift}(\text{sentby}(\theta, p_0).\,\text{args}, \text{sentby}(\theta, p_0)_0.\,\text{args}).$$

□

6.7 Related Work

The method described in this paper represents an extension of the method described by Jonsson in [76] on asynchronous buffered communication to object creation and synchronous message passing in the context of a rendez-vous.

An early reference to some of the techniques we use is Ole-Johan Dahl's "Can Program Proving be Made Practical" (cf. [38]), because this and his work use finite traces for object specification and induction as the prime proof method.

Compatibility, as used in Section 6.4, was introduced by Soundararajan in [144]. In this paper, the existence of a global trace satisfying the interface specification expresses that two objects are compatible, that is, can be composed. The existence of such a trace is not mandatory in our setting, because the global property is proved for all global traces, and is vacuously true, if no global trace satisfying all interface specifications exist.

Instead of predicates in a trace logic, interfaces can also be expressed in temporal logic, as proposed by Lamport in [89], or finite automata, as proposed by Alfaro et al in [41]. Wolfgang Thomas [147] proves that a language based on first order logic is more expressive than a language based on finite automata. For example, a stack discipline, as it occurs in call-return pairs of operation calls, cannot be expressed in this automata setting.

The method presented in this thesis can straightforwardly by applied to the extension of OCL defined in Chapter 5. Known approaches towards tool-based (automatic) verification for object creation concern bounded creation of objects and are implementation dependent, that is, not compositional, as demonstrated by the work of Damm et al [39] and of Ober et al [102].

6.8 Conclusion and Future Work

In this paper we have presented a trace-based specification method for object-oriented programs with object creation. We have proved that not observing object creation but inferring object creation from the trace is a feasible abstraction. Most behavioural specifications are not necessarily concerned with object creation but with the exchange of messages.

We have made explicit that a specification of the behaviour of an object does not only involve the services it provides, but also the context in which this object operates, and the services it requires. Besides this static information, the specification of an object is a contract or a protocol of the behaviour between objects. These are expressed by interface invariants.

We have proposed a proof rule for composing interface specifications and deriving global properties. This allows the verification of systems during early stages of design, where an implementation of each object or class is not yet known.

We have applied our trace logic to a case study: the sieve of Eratosthenes. Our method seems to be particularly well suited for stream-processing applications, as demonstrated by our case study.

Chapter 7

Compositional Verification of Timed Components in PVS

This chapter contains the compositional modelling and verification of the error logic part of the MARS case study [117] in PVS, based on Omega Deliverable 3.3 Appendix 42. We also extend the theory presented in Chapter 6 with real-time.

Essentially, the theory used in this chapter is based on the theory presented in [65]. This theory has been adapted to the needs of object-oriented programming, as presented in the preceding chapters, to deal with timed systems.

One crucial observation is that in the presence of time every event has to be observed when it occurs, and not, like we have done with object creation in Chapter 6, later. The reason is that we also observe the time when some event occurs.

7.1 Introduction

In recent years, UML [111] has been applied to the development of reactive safety-critical systems, in which the quality of the developed software is a key factor. Within the Omega project [116] we have developed a method for the correct development of real-time embedded systems using a subset of UML, which consists of state machines, class diagrams, and object diagrams. In this paper we present a general framework to support compositional verification of such designs defined in this subset of UML using the interactive theorem prover PVS [121, 122]. The framework is based on timed traces. These are an abstraction of the timed semantics of UML state machines as described in [149]. The focus is on the level of components and their interface specifications, without knowing their implementation [44, 66].

Our specifications are based on assertions on timed traces, that is, logical formulae that express the desired properties of a system or one of its components. To be able to formalise intermediate stages during the top-down design of a system we aim at a mixed formalism, that is, a formalism where specifications and programming constructs can be mixed freely. In this paper, we restrict ourselves to parallel composition and hiding. This is inspired by similar work on untimed systems [114, 115, 158] and related to work on timed systems [65, 139].

We apply these general theories to a case study, namely, part of the *Medium Altitude Reconnaissance System* (MARS) deployed by the Royal Netherlands Air Force on the F-16 aircraft [117]. The system employs two cameras to capture high-resolution images. It counteracts image quality degradation caused by the forward motion of an aircraft by creating a compensating motion of the film during the film exposure. The controls applied to the camera for the film speed of the *forward motion compensation* and the frame rate are being computed in real-time based on the current aircraft altitude, ground speed, and some additional parameters. The system is also responsible for producing the frame annotation containing time and the aircraft's current position, which must be synchronised with the film motion. Finally, the system performs health monitoring and alarm processing functions. The part of this case study we focus on the *data-bus manager*. It receives messages from sensors measuring the altitude and its position and tries to identify whether the sensors have broken down, and if they have, whether they have recovered.

This paper is structured as follows. In the next section we describe the semantics of our assertion language. Section 7.3 defines our proof rules. Section 7.4 describes the overall behaviour of our case study. Section 7.5 describes the decomposition of this overall specification into suitable components. Section 7.6 gives a high-level view how the correctness of the decomposition is proved. For the actual proof see [87]. The final Section 7.7 contains some concluding remarks, especially regarding the arduous path leading up to our presented results.

7.2 Semantics

Assertions are predicates on traces θ consisting of observations o. For each such observation we observe the *event* that is occurring, written as $E(o)$, and the time at which it occurs, written as $T(o)$. Time is defined to be a non-negative real, and delays are assumed to be positive. The special event ϵ represents either that time elapses or that some hidden event is occurring.

These traces have to satisfy the following properties in order to be *well-formed*:

1. Time is monotone: $\forall i, j : i \leq j \implies T(\theta_i) \leq T(\theta_j)$.
2. Time progresses, that is, is non-Zeno: $\forall i, \delta : \exists j : i \leq j \wedge T(\theta_i) + \delta \leq T(\theta_j)$.
3. Proper events are instantaneous: $\forall i : E(\theta_i) \neq \epsilon \implies T(\theta_i) = T(\theta_{i+1})$.

Next, we define the projection of a trace θ on a set of events E:

$$\theta \downarrow E \stackrel{\text{def}}{=} \lambda k : \begin{cases} \theta_k, & \text{if } E(\theta_k) \in E \\ \epsilon, & \text{otherwise.} \end{cases}$$

Observe that this projection operator is idempotent, that is, $(\theta \downarrow E) \downarrow E = \theta \downarrow E$ and satisfies: $\forall \theta, E_1, E_2 : \theta \downarrow E_1 = \theta \wedge E_1 \subseteq E_2 \implies \theta \downarrow E_2 = \theta$.

Next we define the notion of a component. A component specifies a set of events E as its signature, that is, as a set of events which a component is able to observe and react to. Usually, these events will be the receiving and sending of messages. As its behaviour the component specifies a set of traces, formalised by a predicate Θ on traces θ over its signature. A component C is defined to be the pair (E, Θ). For any component $C = (E, \Theta)$, we require that its behaviour respects its interface: $\forall \theta : \Theta(\theta) \implies \theta \downarrow E = \theta$.

We define the parallel composition of components $C_1 = (E_1, \Theta_1)$ and $C_2 = (E_2, \Theta_2)$ as $C_1 \parallel C_2 \stackrel{\text{def}}{=} (E_1 \cup E_2, \{\theta \mid \theta \downarrow E_1 \in \Theta_1 \wedge \theta \downarrow E_2 \in \Theta_2 \wedge \theta \downarrow (E_1 \cup E_2) = \theta\})$. That is, the parallel composition of two components maintains the behaviour of its parts, the components synchronise on their common events, and it does not include new events outside the common signature, as in [45, Section 7.4].

For a component $C = (E, \Theta)$ and a set of events E the hiding operator $C - E'$ removes the events in E' from the signature of C. It is formally defined by: $C - E' \stackrel{\text{def}}{=} (E \setminus E', \{\theta \mid \exists \theta' \in \Theta : \theta = \theta' \downarrow (E \setminus E')\})$.

The behaviour of a component is specified by *assertions*, which are predicates on traces. We lift the boolean connectives to assertions in the usual manner.

We define a few suitable abbreviations. The term $E(\theta_i) = e$ states that the event e occurs at position i in the trace θ. The term

$$\text{Never}(e, i, j)(\theta) \stackrel{\text{def}}{=} \forall k : i \leq k \wedge k \leq j \implies E(\theta_k) \neq e$$

asserts that the event e does not occur between positions i and j in the trace θ. Similarly, the assertion

$$\text{Never}(e)(\theta) \stackrel{\text{def}}{=} \forall k : E(\theta_k) \neq e$$

asserts, that e never occurs in a trace. Finally,

$$\text{AfterWithin}(e, i, \delta)(\theta) \stackrel{\text{def}}{=} \exists j : j \geq i \wedge E(\theta_j) = e \wedge T(\theta_j) - T(\theta_i) \leq \delta$$

is an assertion that states that the event e occurs at some position j after i which is no later that δ time units from i.

Specifications of components consist of a signature and an assertion. Because we aim at a mixed framework, in which specifications and programming constructs can be mixed freely, a specification is also considered to be a component. Therefore a specification $S = (E, \Theta)$ is identified by the component $(E, \{\theta \mid \theta \downarrow E = \theta \wedge \Theta(\theta)\})$, which has the same name.

A component $C_1 = (E_1, \Theta_1)$ refines another component $C_2 = (E_2, \Theta_2)$, written $C_1 \implies C_2$, if $E_1 = E_2 \wedge \forall \theta : \Theta_1(\theta) \implies \Theta_2(\theta)$. The refinement relation is a partial order on components and specifications.

We have chosen to use the implication symbol $\implies$ for refinement, because in out theory, refinement (almost) *is* implication. The crucial part of our definition of refinement of components is $\Theta_1(\theta) \implies \Theta_2(\theta)$! The same idea also holds for Lamport's TLA [89].

7.3 Compositional Proof Rules

In this section we derive a number of compositional proof rules. Their correctness is checked in PVS based on the semantic definitions and the definition of specifications [87]. We start with a consequence rule, which allows the weakening of assertions in specifications.

Let $C_1 = (E_1, \Theta_1)$ and $C_2 = (E_2, \Theta_2)$ be two specifications. Then

$$(E_1 = E_2 \wedge (\forall \theta : \Theta_1(\theta) \implies \Theta_2(\theta))) \implies (C_1 \implies C_2) \quad .$$

To define a sound rule for parallel composition, we first show that the validity of an assertion Θ only depends on its signature. This is specified using the following predicate:

$$\text{depends}(\Theta, E) \stackrel{\text{def}}{\iff} \forall \theta, \theta' : \Theta(\theta) \wedge \theta \downarrow E = \theta' \downarrow E \implies \Theta(\theta') \quad .$$

Then we can establish $\forall E : \text{depends}(\Theta, E) \iff (\forall \theta : \Theta(\theta) \iff \Theta(\theta \downarrow E))$. Using this statement we can prove the soundness of the parallel composition rule:

$$(\text{depends}(\Theta_1, E_1) \wedge \text{depends}(\Theta_2, E_2)) \implies ((E_1, \Theta_1) \parallel (E_2, \Theta_2) \implies (E_1 \cup E_2, \Theta_1 \wedge \Theta_2)) \quad .$$

To be able to use refinement in a context, we derive a monotonicity rule:

$$((C_1 \implies C_2) \wedge (C_3 \implies C_4)) \implies ((C_1 \parallel C_3) \implies (C_2 \parallel C_4)) \quad .$$

Similarly, we prove a compositional rule and a monotonicity rule for the hiding operator.

$$\text{depends}(\Theta, E_1 \setminus E_2) \implies (((E_1, \Theta) - E_2) \implies (E_1 \setminus E_2, \Theta))$$
$$(C_1 \implies C_2) \implies ((C_1 - E) \implies (C_2 - E)) \quad .$$

7.4 The MARS Example

From the MARS example we consider only a small part, namely the *data bus manager*. This part serves as an illustration on how to apply the presented techniques to a timed system. Figure 7.1 shows the architecture of the data bus manager.

The external data sources *altitude data source* and a *navigation data source* send data, here represented by abstract events d_1 and d_2, respectively, to a *message receiver*. If the sources function correctly, they send data with period P and jitter $J < \frac{1}{2}P$, as depicted in Figure 7.2: Data should be available during the grey periods.

For any data source s its behaviour can be specified by the assertion $\text{DS}_{s,1}(\theta) \wedge \text{DS}_{s,2}(\theta)$ on its traces of observations θ, where $\text{DS}_{s,1}$ and $\text{DS}_{s,2}$ are defined below. The

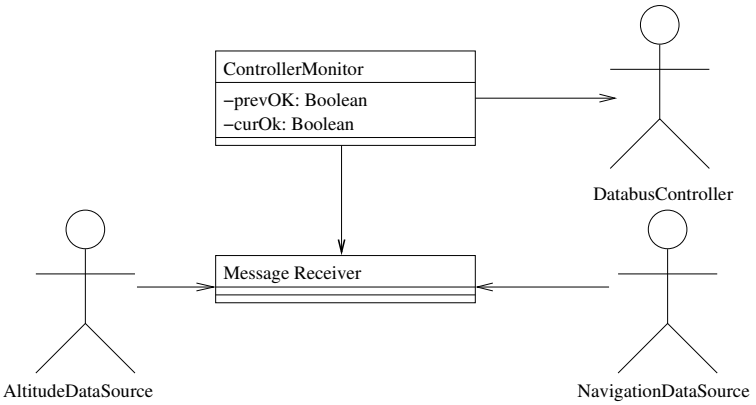


Figure 7.1: Architecture of the data bus manager

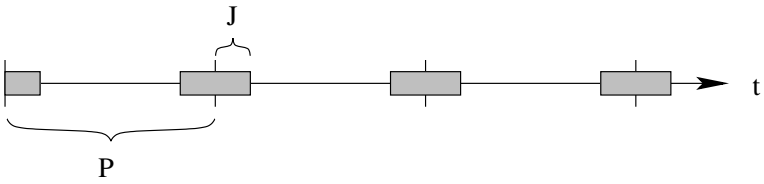


Figure 7.2: Data with period P and jitter J

assertion $\mathrm{DS}_{s,1}$, where s ranges over $1, 2$, specifies that each occurrence of an event d_s is within the period specified by the jitter. The assertion $\mathrm{DS}_{s,2}$ specifies that at most one such message is sent during this period:

$$\mathrm{DS}_{s,1}(\theta) \overset{\text{def}}{\iff} \forall i : E(\theta_i) = d_s \implies \exists n : nP - J \leq T(\theta_i) \wedge T(\theta_i) \leq nP + J$$

$$\mathrm{DS}_{s,2}(\theta) \overset{\text{def}}{\iff} \forall i, j : E(\theta_i) = d_s \wedge E(\theta_j) = d_s \implies i = j \vee P - 2J \leq |T(\theta_i) - T(\theta_j)| \quad .$$

Consequently, a data source will not send data outside of the assigned time frame and will also not send more than one data sample during this time frame. A state machine depicting such normal behaviour is shown in Figure 7.3. The data source may send data only while it is in the state Initial or in the state Send. While it is in the Initial state, it may stay in this state for at most J time units.

If a data source fails to send a data item for K consecutive times, then the bus manager shall indicate the error by sending an *err* signal. That this situation has occurred is formalised by an appropriate timeout assertion:

$$\mathrm{TimeOut}(e, t, i, j)(\theta) \overset{\text{def}}{\iff} \mathrm{Never}(e, i, j) \wedge T(\theta_j) - T(\theta_i) > t$$

This assertion states that the event e has not occurred for at least t time units between positions i and j of a trace θ.

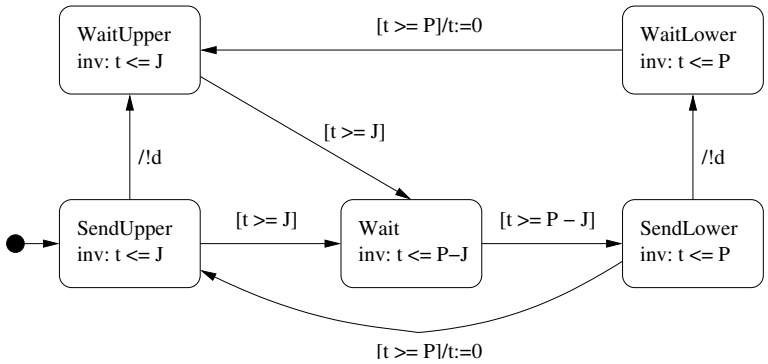


Figure 7.3: State machine of the data source

The system is said to have recovered, if N consecutive data messages have been received from each source. The occurrence of N consecutive events e between i and j is specified by the predicate $\mathrm{occ}(e, N, i, j)$, which is defined as:

$$\begin{aligned}\mathrm{occ}(e, N, i, j)(\theta) \overset{\text{def}}{\iff} & \; N = 0 \vee \exists f : |\operatorname{dom}(f)| = N \wedge f(0) = i \wedge \\ & f(|\operatorname{dom}(f)| - 1) = j \wedge (\forall k : k \leq |\operatorname{dom}(f)| - 1 \implies E(\theta_{f(k)}) = e) \wedge \\ & (\forall k : k < |\operatorname{dom}(f)| - 1 \implies f(k) < f(k+1) \wedge \\ & P - J < T(\theta_{f(k+1)}) - T(\theta_{f(k)}) \wedge T(\theta_{f(k+1)}) - T(\theta_{f(k)}) < P + J) \quad .\end{aligned}$$

This implies that there exists a strictly monotonically increasing sequence f of length N of indexes starting at i and ending at j such that at each position in this sequence the event e occurs and that these events occur $P \pm J$ time-units apart.

We can now define that a data source s is in an error state at position i in the trace θ by observing that it has not sent data for at least $L \overset{\text{def}}{=} KP + 2J$ time units at position $j \leq i$ and that it has not recovered until position i:

$$\begin{aligned}\mathrm{Error}(d, i)(\theta) \overset{\text{def}}{\iff} & \; \exists k, j : j \leq i \wedge \mathrm{TimeOut}(d, L, k, j)(\theta) \wedge \\ & (\forall m : j < m \wedge m \leq i \implies \neg \exists l : \mathrm{occ}(d, N, l, m)(\theta))\end{aligned}$$

The validity of an error signal is specified by the following predicates:

$$\begin{aligned}\mathrm{TDS}_1(\theta) \overset{\text{def}}{\iff} & \; (\forall i, j : i < j \wedge (\exists s : \mathrm{TimeOut}(d_s, L, i, j)(\theta)) \wedge \\ & (\forall s : \neg \mathrm{Error}(d_s, j)(\theta))) \implies \mathrm{AfterWithin}(err, j, \Delta_{err})(\theta) \quad .\end{aligned}$$

Here, Δ_{err} expresses a delay that models the time the system needs to react to the

occurrence of an error. The integrity of the error signal *err* is specified by:

$$\begin{aligned}\mathrm{TDS}_2(\theta) \overset{\text{def}}{\iff} \forall j : E(\theta_j) = err \implies \exists i,k : i < k \wedge k < j \wedge \\ (\exists s : \mathrm{TimeOut}(d_s, L, i, k)(\theta)) \wedge (\forall s : \neg\, \mathrm{Error}(d_s, k)(\theta)) \wedge \\ \mathrm{Never}(err, k, j-1)(\theta) \quad .\end{aligned}$$

The system recovers from an error when all data sources have been sending N consecutive messages. This recovery is indicated by sending a *ok* signal. The next predicate specifies that all sources have indeed sent N consecutive data messages from $D = \{d_s \mid s \in S\}$:

$$\begin{aligned}\mathrm{Recover}(D, i, j) \overset{\text{def}}{\iff} \exists f, g : i = \min_{d \in D} f(d) \wedge j = \max_{d \in D} g(d) \wedge \\ (\forall d, d' : |T(\theta_{f(d)}) - T(\theta_{f(d')})| \leq 2J) \wedge \\ (\forall d, d' : |T(\theta_{g(d)}) - T(\theta_{g(d')})| \leq 2J) \wedge \\ (\forall d : \mathrm{occ}(d, N, f(d), g(d))) \quad .\end{aligned}$$

This predicate states that there exist two functions f and g from events to positions such that i is the smallest value produced by f, j is the largest value produced by g, the values in the range of f are at most $2J$ time units apart, as are the values in the range of g such that we have N occurrences of d between $f(d)$ and $g(d)$. Using this predicate, we can define the validity of the *ok* signal:

$$\begin{aligned}\mathrm{TDS}_3(\theta) \overset{\text{def}}{\iff} \forall i, j : i < j \wedge \mathrm{Recover}(\{d_s \mid s \in S\}, i, j)(\theta) \wedge \\ (\exists s : \mathrm{Error}(d_s, j)(\theta)) \implies \mathrm{AfterWithin}(ok, j\Delta_{ok})(\theta) \quad .\end{aligned}$$

Similar to Δ_{err}, the delay Δ_{ok} models the time needed by the error logic to react to the recovery of the system. The integrity of the *ok* signal is specified by:

$$\begin{aligned}\mathrm{TDS}_4(\theta) \overset{\text{def}}{\iff} \forall j : E(\theta_j) = ok \implies \exists i,k : i < k \wedge k < j \wedge (\exists s : \mathrm{Error}(d_s, i)(\theta)) \wedge \\ \mathrm{Recover}(\{d_s \mid s \in S\}, i, k)(\theta) \wedge \mathrm{Never}(ok, k, j-1)(\theta) \quad .\end{aligned}$$

Finally, we specify the behaviour of the global system by the assertion TDS:

$$\mathrm{TDS}(\theta) \overset{\text{def}}{\iff} \mathrm{TDS}_1(\theta) \wedge \mathrm{TDS}_2(\theta) \wedge \mathrm{TDS}_3(\theta) \wedge \mathrm{TDS}_4(\theta) \quad .$$

From this global specification we derive a version of the data bus manager in such a way that full compositional deductive verification becomes possible and the verification task is split into smaller verification tasks.

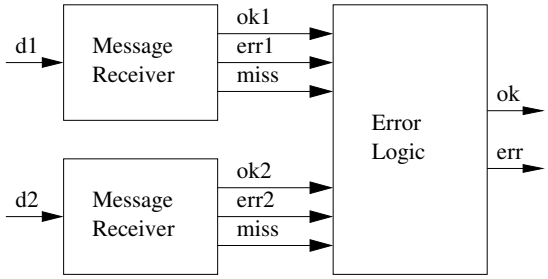


Figure 7.4: Decomposed architecture for two data sources

7.5 Decomposition of the MARS example

The main idea is that we specify a separate data receiver for each data type d and later compose the receivers for different data sources with a component that specifies the combinations of errors and recovery. This architecture is depicted in Figure 7.4.

The *message receivers* are two identical processes, whose internal states are made visible by external signals *err*, *miss*, and *ok* to represent error and recovery. Hence the message receiver is specified by a component, parameterised over events d, *err*, *miss*, and *ok*. The role of *miss* signals will be explained later.

7.5.1 Message Receiver

The message receiver processes the data received from one data source. Processing data takes some time, which varies depending on the data received. We assume that this time is between l and u. The message receiver should enter an error state if if K, say 3, successive messages are missing from its source. It should resume normal operation if it has received N, say 2, successive messages from its source. This behaviour is depicted by the state machine in Figure 7.5.

The message receiver receives data from one data source and counts how many consecutive messages have been absent from the source. We can assert that an error is present once no message has been received since L time units. Recall, that this is the time required to observe that K messages have been absent from the input stream. If this occurs, the message receiver sends a err_s message to an error logic component to indicate to it that from one source K messages have been missing. The delay $\Delta^{\mathrm{MR}}_{err}$ models the time needed by the message receiver to send its err_s signal. This is specified by:

$$\mathrm{MR}_{s,1}(\theta) \stackrel{\mathrm{def}}{\Longleftrightarrow} \forall i, j : \mathrm{TimeOut}(d_s, L, i, j)(\theta) \wedge \neg\, \mathrm{Error}(d_s, i)(\theta) \implies \mathrm{AfterWithin}(err_s, j, \Delta^{\mathrm{MR}}_{err})(\theta) \quad .$$

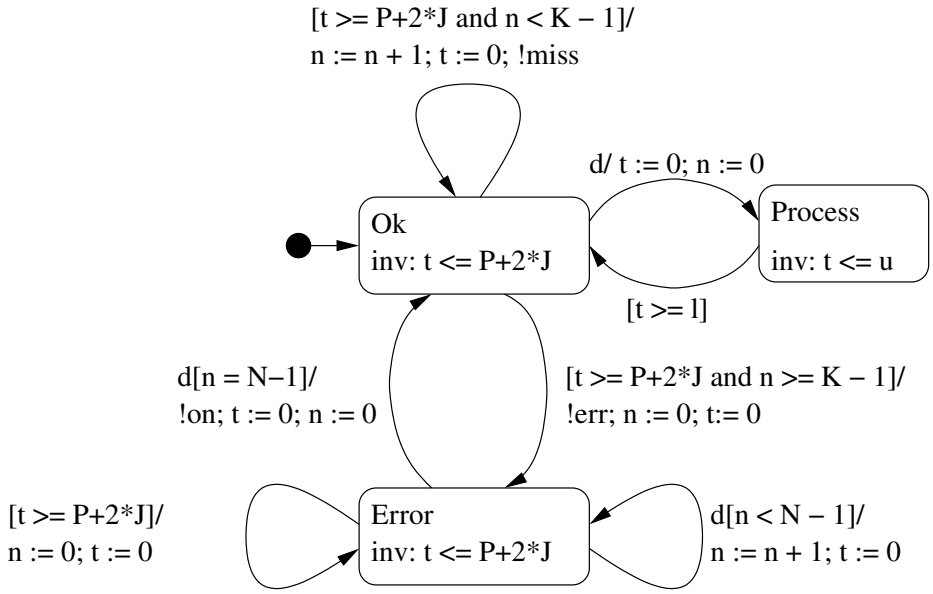


Figure 7.5: State machine of the message receiver

The next predicate specifies that the message receiver will only send an error signal e if a timeout has occurred:

$$\mathrm{MR}_{s,2}(\theta) \overset{\text{def}}{\iff} \forall j : E(\theta_j) = err_s \implies \exists i,k : i < k \wedge k < j \wedge \\ \neg\,\mathrm{Error}(d_s, i)(\theta) \wedge \mathrm{TimeOut}(d_s, L, i, j)(\theta) \wedge \mathrm{Never}(err_s, k, j-1)(\theta) \quad .$$

Next, we define the validity and integrity of an *ok* signal. If the message receiver is in error state and it receives N consecutive d messages, then it will send an *ok* message within $\Delta_{ok}^{\mathrm{MR}}$ time units:

$$\mathrm{MR}_{s,3}(\theta) \overset{\text{def}}{\iff} \forall j : \mathrm{Error}(d_s, j)(\theta) \wedge \mathrm{Recover}(d_s, j)(\theta) \implies \\ \mathrm{AfterWithin}(ok_s, j, \Delta_{ok}^{\mathrm{MR}})(\theta) \quad .$$

The next predicate specifies that if an *ok* signal occurred, then the system was in an error state before and N consecutive data messages had occurred, and no other *ok* signal has been emitted in between:

$$\mathrm{MR}_{s,4}(\theta) \overset{\text{def}}{\iff} \forall j : E(\theta_j) = ok_s \implies \exists i,k : i < k \wedge k < j \wedge \\ \mathrm{Error}(d_s, i)(\theta) \wedge \mathrm{occ}(d_s, N, i, k) \wedge \mathrm{Never}(ok_s, k, j-1)(\theta) \quad .$$

Next, the error logic component, to be specified later, has to be notified by any message receiver that did not receive a data message in time. This is indicated by a

miss message. We introduce this *miss* signal, because using only *err* and *ok* signals is not sufficient for recovery according to the specification. The problem is that the *err* signal indicates the absence of K data items, whereas recovery requires the presence of N consecutive data signals from the data source. Observe that, when staying in the correct operational mode, a few missing data items are allowed, but this is not allowed when trying to recover.

Only *one miss* signal is needed and not one per message receiver. The reason for this is that, in order to recover, *all* message receivers have to receive N consecutive data messages. The presence of a *miss* signal indicates that there is one component which missed a data message during this period. Consequently, the error logic does not need to know which message receiver missed the signal.

A message receiver sends a *miss* message to the error logic whenever a data message has timed out from the data source *and* it is not in an error state. Again, sending a *miss* signal is delayed by $\Delta^{\mathrm{MR}}_{miss}$ time units.

$$\mathrm{MR}_{s,5}(\theta) \overset{\text{def}}{\iff} \forall j : \mathrm{TimeOut}(d_s, P + 2J, i, j)(\theta) \wedge \neg\, \mathrm{Error}(d_s, j)(\theta) \implies \mathrm{AfterWithin}(\mathit{miss}, j, \Delta^{\mathrm{MR}}_{\mathit{miss}})(\theta)$$

If the message receiver is already in an error state, it will signal N consecutive data messages using an *ok* message. Therefore, sending the *miss* signal is not necessary in this case. Also note that if we miss the Kth data item at j, the $\mathrm{Error}(d_s, j)(\theta)$ predicate is true and no *miss* signal is emitted. Instead, following the assertion $\mathrm{MR}_{s,3}$, an err_s signal will be sent. That is, it will not send both a *miss* signal and an err_s signal.

The next predicate specifies the integrity of a *miss* event, that is, whenever a *miss* signal occurs the system was not in an error state before the occurrence of the *miss* signal, a data messages has not been received within the specified time, and no other *miss* signal has been emitted between the time-out of the data message and the occurrence of the *miss* signal.

$$\mathrm{MR}_{s,6}(\theta) \overset{\text{def}}{\iff} \forall j : E(\theta_j) = \mathit{miss} \implies \exists i, k : i < k \wedge k < j \wedge \neg\, \mathrm{Error}(d_s, i)(\theta) \wedge \mathrm{TimeOut}(d_s, P + 2J, i, k)(\theta) \wedge \mathrm{Never}(\mathit{miss}, k, j - 1)(\theta)$$

From N missing *miss* signals we can conclude, that the data source s has received N consecutive data messages:

Lemma 7.1. *For any message receiver s we have: if* $\forall i, j : \mathrm{TimeOut}(\mathit{miss}, NP + 2J, i, j)$ *then* $\mathrm{occ}(d_s, N, i, j)$.

More importantly, the timeout of the *miss* signal implies that all message receivers have received N consecutive data messages.

Corollary 7.2. *If* $\mathrm{TimeOut}(\mathit{miss}, NP + 2J, i, j)$ *for all* i, j, *then* $\mathrm{Recover}(\{d_s \mid s \in S\}, i, j)$.

Proof. Follows directly from Lemma 7.1 and the definition of Recover [87]. □

Finally, we specify a message receiver for a source s as

$$\mathrm{MR}_s(\theta) \overset{\text{def}}{\iff} \mathrm{MR}_{s,1}(\theta) \wedge \mathrm{MR}_{s,2}(\theta) \wedge \mathrm{MR}_{s,3}(\theta) \wedge \mathrm{MR}_{s,4}(\theta) \wedge \mathrm{MR}_{s,5}(\theta) \wedge \mathrm{MR}_{s,6}(\theta) \quad .$$

In the following section we formalise the interface specification of the error logic.

7.5.2 Error Logic

The error logic component accepts err_s and ok_s signals from each data source s. It also accepts a signal *miss* which indicates that there exists a data source s that has not received data from its source during its cycle. The error logic will emit an *err* signal if it detects an error in the system and an *ok* signal if the system recovers after an error. The behaviour of the error logic is specified in the state machine of Figure 7.6.

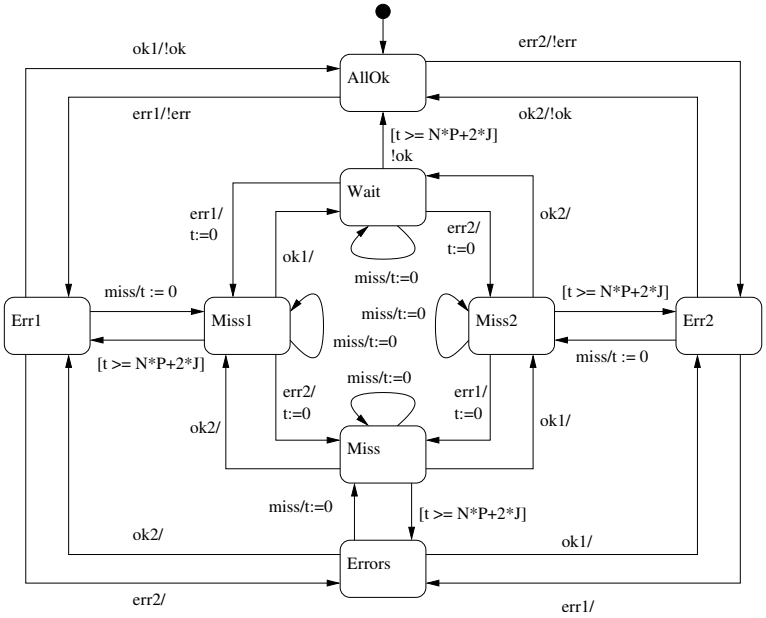


Figure 7.6: State Machine of the error logic component

In this state machine, the state AllOk indicates that the system is in the mode of normal operation. If the system is in this state and receives an err_i signal from the message receiver i, then it moves to the state err_i and signals an error by sending an *err* signal. The system may recover if it receives an ok_i signal or it may receive an err_j signal for $i \neq j$. That the system has to recover from an error from both data sources is indicated by staying in the state Errors. Moving to Errors does not require sending an

err signal, because we have already done so by taking a transition from AllOk to the state err_i for some i. The system signals its recovery by sending an *ok* signal if it has received an ok_i signal while in the state err_i.

As long as the system is in the AllOk state it ignores all *miss* signals. If a data source i failed to send K consecutive data messages to its message receiver, this will be signalled by the message receiver with a err_i signal.

If the system is in an error state, that is, in one of the states err_1, err_2, or Errors, and it receives a *miss* signal, the error logic has to record that it has to wait for a time out of the miss signal and the required number of *ok* signals in order to return to normal operation. This fact is recorded in either the state $miss_1$, $miss_2$, *miss*, or Wait. These states indicate that the system has to wait for a time-out of the *miss* signal and for either an ok_1, ok_2, both, or no *ok* signal from the message receivers in order to recover. The state Wait is entered, whenever one of the $miss_i$ states has been left after receiving the corresponding ok_i signal, and the error logic itself has to emit an *ok* signal to confirm that the system has recovered from the error condition. Observe that we have to wait until the end of the current period in order to assert that during this time neither message receiver sends an error signal. While staying in one of these states, the system measures the time elapsed since the latest reception of a *miss* signal using the clock t. Therefore it always resets t if it receives a *miss* signal *or* a err_i signal. Recall, that the message receiver will not send both a *miss* signal and an err_i signal during the same cycle.

The Wait state moves to the AllOk state after a timeout of the *miss* signal by emitting an *ok* signal. Note that the Wait signal is reachable by the following scenario: The system starts in the AllOk state. It then receives an err_1 signal from message receiver 1. Having received this signal the state changes to err_1. During the next period it receives a *miss* signal from message receiver 2! This causes a change of the state to $miss_1$, indicating that it has to receive an ok_1 signal from message receiver 1 and has to wait that message receiver 2 has to receive N consecutive data messages. Observe that in this situation message receiver 1 only has to receive $N-1$ data messages. Assuming that both message receivers will receive their data messages, message receiver 1 sends its ok_1 signal after $N-1$ periods, after which the error logic changes its state to Wait. Now the error logic has to wait another period in order to assert that message receiver 2 has received its Nth data message, after which it may signal recovery.

We proceed by formalising the behaviour of the error logic component. The error logic component accepts the signals err_1, err_2, *miss*, ok_1, and ok_2 as inputs and sends the signals *err* and *ok* as outputs.

Whether the error logic knows that a source s is in an error state at position i of trace θ is indicated by the following predicate:

$$\mathrm{Error}(i, s)(\theta) \overset{\text{def}}{\iff} \exists m : m \leq i \wedge E(\theta_m) = err_s \wedge \mathrm{Never}(ok_s, m+1, i)(\theta) \quad .$$

Whether the error logic is in an AllOk state at position i of a trace can be computed

by the following predicate:

$$\text{AllOk}(i)(\theta) \overset{\text{def}}{\iff} \forall s : \neg\, \text{Error}(i, s)(\theta) \quad .$$

Sending the *err* signal after having received an err_s signal from some message receiver s is delayed by Δ_{err}^{EL} time units. Then the validity of an *err* signal indicating error is specified by:

$$\text{EL}_1(\theta) \overset{\text{def}}{\iff} \forall i : \text{AllOk}(i)(\theta) \wedge (\exists s : E(\theta_{i+1}) = err_s) \implies \text{AfterWithin}(err, i+1, \Delta_{err}^{\text{EL}}) \quad .$$

Its integrity is specified by:

$$\text{EL}_2(\theta) \overset{\text{def}}{\iff} \forall j : E(\theta_j) = err \implies \exists i : i < j \wedge \text{AllOk}(i)(\theta) \wedge (\exists s : (E(\theta_{i+1}) = err_s) \wedge \text{Never}(err, i+2, j-1)(\theta) \quad .$$

The next predicate states, that a data source s recovers from an error:

$$\text{Recover}(i, s)(\theta) \overset{\text{def}}{\iff} \forall i : \text{Error}(i-1, s)(\theta) \wedge E(\theta_i) = ok_s \quad .$$

The validity of an *ok* signal indicating recovery of a component s is specified by, where emitting the *ok* signal is delayed by Δ_{ok}^{EL} time units:

$$\text{EL}_3(\theta) \overset{\text{def}}{\iff} \forall i : \exists s : \text{Recover}(i, s)(\theta) \wedge (\forall s : \neg\, \text{Error}(i, s')) \wedge (\exists k : \text{TimeOut}(miss, NP + 2J, k, i)(\theta)) \implies \text{AfterWithin}(ok, i, \Delta_{ok}^{\text{EL}}) \quad .$$

Observe that the assertion $\text{Recover}(i, s)(\theta) \wedge (\forall s' : \neg\, \text{Error}(i, s'))$ states that the message receiver s is the last message receiver to recover.

The integrity of the *ok* signal is specified by:

$$\text{EL}_4(\theta) \overset{\text{def}}{\iff} \forall j : E(\theta_i) = ok \implies \exists i : i < j \wedge (\exists s : \text{Recover}(i, s)(\theta)) \wedge (\forall s : \neg\, \text{Error}(i, s)) \wedge \text{Never}(ok, i+1, j-1) \wedge (\exists k : \text{TimeOut}(miss, NP + 2J, k, i)(\theta)) \quad .$$

The error logic is then specified by the assertion:

$$\text{EL}(\theta) \overset{\text{def}}{\iff} \text{EL}_1(\theta) \wedge \text{EL}_2(\theta) \wedge \text{EL}_3(\theta) \wedge \text{EL}_4(\theta) \quad .$$

7.6 Correctness of the decomposition

To show the correctness of the decomposition of the global system into a receiver for each data source and an error logic, we first define general signals with a source identity and process identities S.

We observe that the internal signal are $I \stackrel{\text{def}}{=} \{ok_s, err_s, miss \mid s \in S\}$ and that the externally observable signals are $\{d_s, err, ok \mid s \in P\}$. The statement of correctness then is:

Theorem 7.3. $((\parallel_{s\in S} \mathrm{MR}_s) \parallel \mathrm{EL}) - I \implies \mathrm{TDS}$.

To establish this theorem, we first need to establish a sequence of intermediate steps, where we established the validity of each part of the TDS assertion individually. The proofs in PVS can be found in [87].

Lemma 7.4. $\Delta^{\mathrm{MR}}_{err} + \Delta^{\mathrm{EL}}_{err} \leq \Delta_{err} \wedge (\forall s : \mathrm{MR}_{s,1}(\theta)) \wedge \mathrm{EL}_1(\theta) \implies \mathrm{TDS}_1(\theta)$

The proof of this lemma is straight forward but tedious. The condition $\Delta^{\mathrm{MR}}_{err} + \Delta^{\mathrm{EL}}_{err} \leq \Delta_{err}$ is needed to establish that the composed system sends the *err* signal in time.

Lemma 7.5. $\Delta^{\mathrm{MR}}_{err} + \Delta^{\mathrm{EL}}_{err} \leq \Delta_{err} \wedge (\forall s : \mathrm{MR}_{s,2}(\theta)) \wedge \mathrm{EL}_2(\theta) \implies \mathrm{TDS}_2(\theta)$

Lemma 7.6. $\Delta^{\mathrm{MR}}_{ok} + \Delta^{\mathrm{EL}}_{ok} \leq \Delta_{ok} \wedge (\forall s : \mathrm{MR}_{s,3}(\theta)) \wedge \mathrm{MR}_{s,5}(\theta)) \wedge \mathrm{EL}_3(\theta) \implies \mathrm{TDS}_3(\theta)$

In the proof of this lemma we use Lemma 7.1 in conjunction with the assumption $\forall s : \mathrm{MR}_{s,5}(\theta)$ to establish that each data source has N occurrences of a data signal from each component. The assumption $\forall s : \mathrm{MR}_{s,3}(\theta)$ asserts that a valid ok_s signal is sent. The condition $\Delta^{\mathrm{MR}}_{ok} + \Delta^{\mathrm{EL}}_{ok} \leq \Delta_{ok}$ is needed to establish that the *ok* signal occurs in time.

Lemma 7.7. $\Delta^{\mathrm{MR}}_{ok} + \Delta^{\mathrm{EL}}_{ok} \leq \Delta_{ok} \wedge (\forall s : \mathrm{MR}_{s,4}(\theta)) \wedge \mathrm{MR}_{s,6}(\theta)) \wedge \mathrm{EL}_4(\theta) \implies \mathrm{TDS}_4(\theta)$

7.7 Conclusions

We have presented a compositional framework for the compositional verification of high-level real-time components which communicate by means of asynchronous signals. This framework reflects the full formal proof given in PVS [87], and is the result of a long and arduous path leading to consistent specifications of the parts and their properties. Compositional proof rules for parallel composition and hiding have been proved sound in PVS. The framework is based on timed event traces which are abstractions of the runs of the semantics of timed UML state machines [149]. In this way, we can use deductive verification in PVS to prove the correctness of a decomposition of a system into a number of asynchronously communicating components. Next, the components can be implemented independently using UML, according to

their specification, and the correctness of the implementation with respect to the interface specification may be established by means of other techniques, such as model checking.

The framework has been applied to the MARS case study provided by the Netherlands National Aerospace Laboratory. This case study has been supplied in the form of UML models. In general, interactive verification of UML models is very complex because we have to deal with many features simultaneously, such as timing, synchronous operation calls, asynchronous signals, threads of control, and hierarchical state machines. Hence, compositionality and abstraction are essential to improve scalability.

The compositional verification of the case study presented here shows that deductive verification is more suitable for the correctness proofs of high-level decompositions, to eventually obtain relatively small components that are suitable for model checking.

Because we started with a specification, which was monolithic and contained errors, a redesign of the original system was necessary to enable the application of compositional techniques and to help improving our understanding of the model. Interestingly, this led to a design that is more flexible, for example, for changing the error logic, and more easily extensible, for example, to more data sources, than the original model.

We have found errors in the decomposed specification using model checking (we used the IF validation environment [16] and UPPAAL [91]) and by the fact that no proof in PVS could be found for the original specification. One of these errors was that we did not include a *miss* signal, which is needed to correctly observe recovery in the error logic component. This allowed the recovery of the system in circumstances where the global specification did not allow this.

Observe that the compositional approach requires substantial additional effort to obtain appropriate specifications for the components. Finding suitable specifications is difficult. Hence, it is advisable to start with finite high-level components and to simulate and to model-check these as much as possible. Apply interactive verification only when sufficient confidence has been obtained. Finally, it is good to realise that interactive verification is quite time consuming and requires detailed knowledge of the tool.

One final conclusion from this example is that, for specifications of the complexity of the MARS case study, especially with respect to its error logic, confidence in its correct functioning requires tool-based verification.

Chapter 8

Conclusion

In the introduction we have written that we develop a formal semantics for a subset of UML in this dissertation, which allows the formal validation of real-time embedded systems modelled with UML and specified with OCL. To achieve this, we have developed a formal semantics for UML class diagrams, object diagrams, and OCL, suitable for an embedding into the theorem prover PVS. The embedding uses the formal semantics of state machines developed by van der Zwaag and Hooman [149].

8.1 Summary

This dissertation represents a further step towards a rigorous formalisation of UML and towards OCL for tool supported formal verification of models. Our results extend, update, or complement the results obtained by Mark Richter [133] and Demissie Aredo [6] in various directions:

- We have described a formal semantics of UML class diagrams and UML object diagrams. This semantics establishes object diagrams as models of class diagrams and constraints as invariants on object diagrams. Multiplicities of associations are understood as constraints, that is, as invariants of object diagrams.
- In currently available UML case tools even basic support for OCL is missing. Some tool vendors provide plug-ins which validate OCL constraints, but they are mostly used during detailed design; constraints are used in the same way assertions are used in programming languages. One reason for this we have identified is the inflexibility of the type system of OCL. To alleviate this, we have proposed a generalisation of this type system, demonstrated that it offers greater flexibility in developing the underlying model, and implemented a type checker based on this extension.
- OCL is a constantly evolving language. We have provided a formal semantics for OCL 2.0, which we have formalised in the language of the theorem prover PVS. This formalisation is supported by a translator, which translates UML models and OCL constraints into PVS. The translator implements a shallow embedding

of OCL into PVS. The feasibility of this approach has been demonstrated by verifying case studies, among others, the Sieve of Eratosthenes, using this tool.

- We have analysed the newly introduced *OCL Message Expressions*, and found them lacking, because they only allow to specify whether a message has been sent during the execution of an operation call. We have proposed to include histories into OCL, which allow to specify whether messages have been sent *or received* and to reason about the order of these. We have demonstrated that all OCL message expressions can be expressed in terms of OCL constraints on histories. Therefore, histories form a conservative extension of OCL. Histories, however, offer much greater flexibility, in that these histories may also be used in class invariants and preconditions of operations.

- We analysed the implications of object creation for a history-based formalism. This has led to the axiomatisation of histories with object creation, which we have formalised in the PVS theorem prover. We show that in the case of reactive systems, that is, systems in which each action is the reaction to a message received, it is not necessary to observe object creation. A safe approximation of the tree of creation can be computed from histories. If objects are active, we observe *prenatal object activity,* that is, objects initially appear to be root objects (which have not been created by any object) but later one may observe that they have indeed been created by other objects. Using this axiomatisation we have proved a case study, the Sieve of Eratosthenes, correct.

- The final extension of the trace based formalism was to extend our notion of communication traces with time. We have applied this method to the error logic component of the medium altitude reconnaissance system (MARS) in order to derive a correct specification of this component.

8.2 Future Research

Formalising the UML is very challenging, not only because of the sheer size of the standard, its number of concepts and notations. Even the designers of UML have expressed the opinion that there is too much overlap and redundancy in the notations of UML. The reason is that the Object Management Group not only formalises best practises, but also suggests new notations and new technology, which become part of the standards, without having proved their usefulness in practise. It has been suggested by Cris Kobryn, who chaired the standardisation of UML 2.0, that UML 3.0 should only contain the parts of UML 2.0 which have proved their usefulness in practise [80]. Therefore, formalising every aspect of the UML should *not* be a goal for future research.

Instead, one should focus on identifying a *small* selection of notations from UML and develop a formally sound system development method around these notations. The notations we selected for this dissertation are quite powerful for modelling event-driven systems. Architectural diagrams may be added, because they display aspects of class diagrams and object diagrams in an integrated manner.

In this dissertation we have not considered the formalisation of state machines. Instead, we used the semantics proposed by van der Zwaag and Hooman [149]. This semantics differs from the semantics described in the UML standard [111]. A semantics has been derived from the standard by Schönborn [140]. This semantics may be formalised in PVS and used with our tool. This would result in an integrated tool that allows the formal verification of UML 2.0 models in PVS.

The methods described in this dissertations may be extended with refinement-based methods. This extension supports a top-down development of systems driven by formal development steps.

In order to make the UML and the OCL useful, strong semantical analysis tools and validation tools have to be developed. Without these tools, the UML and especially OCL is only a write-only specification formalism, and consequently a weak formal method in the sense of Wolper [156]. A specification is called write-only, if it is only written for the sake of compliance with standardised development methods, but never used for the development of the specified system.

In this dissertation we have analysed the light-weight end of these methods, namely type-checking, and the heavy-weight end, namely interactive theorem proving, without covering the methods in between these extremes. Especially strong static analysis methods have the potential of improving the design of safety critical systems by detecting errors early.

Because the risks involved in designing and implementing embedded real-time systems, which are very often safety critical, strong formal methods are needed, which allow the rigorous analysis and formal verification of models, from which eventually a working system has to be designed. This is one of the directions suggested by Grady Booch [12], who is one of the fathers of UML:

> Note that the OMG speaks of MDA (Model-Driven Architecture). In my world view, the UML can be much more than just a means of constructing systems: for example, consider debugging a system at the level of UML diagrams.

This dissertation provides an approach to verifying UML diagrams, and through this also a way for debugging UML diagrams. But one draw-back of our approach is that it needs a lot of expertise on using the theorem prover *and* the translator in order to trace a detected error back to its location in the original diagram. Therefore, one direction of research is to achieve seamless integration between theorem proving and the original diagram, perhaps by defining a formal deductive system for OCL and build a proof

checker for this deductive system. The major obstacle to overcome is to find deductive rules for late-binding of functions.

Appendix A

OCL 2.0 Grammar

In this appendix we summarise the concrete syntax of OCL [113] using an extended Backus-Naur format [8]. The grammar in [113] is different from the grammar presented here. It is incomplete and context-sensitive parser. The grammar described here is a context-free grammar suitable for predictive parsers with 3 tokens of look-ahead. The start symbols are ⟨file⟩ and ⟨expression⟩. The reserved keywords of OCL are given in Table A.1:

`and`	`context`	`def`	`derive`	`else`
`endif`	`endpackage`	`false`	`if`	`implies`
`in`	`init`	`inv`	`let`	`null`
`not`	`or`	`package`	`post`	`pre`
`then`	`true`	`xor`		

Table A.1: Reserved keywords

A.1 Literals

We define the literals of OCL. All white-space characters are token delimiters.

⟨literal⟩ ::= ⟨primitive-literal⟩ | ⟨collection-literal⟩ | ⟨tuple-literal⟩
⟨primitive-literal⟩ ::= ⟨boolean-literal⟩ | ⟨integer-literal⟩ | ⟨real-literal⟩
| ⟨string-literal⟩ | `null`
⟨boolean-literal⟩ ::= `true` | `false`
⟨integer-literal⟩ ::= `0`..`9`{`0`..`9`}
⟨real-literal⟩ ::= ⟨integer-literal⟩.⟨integer-literal⟩[(`e` | `E`)[−]⟨integer-literal⟩]
⟨string-literal⟩ ::= `'`{⟨characters⟩}`'`
⟨identifier⟩ ::= (⟨letter⟩ | _){⟨letter⟩ | _ | ⟨digit⟩}
⟨path-name⟩ ::= ⟨identifier⟩ | ⟨path-name⟩`::`⟨identifier⟩

⟨collection-literal⟩ ::= ⟨identifier⟩{[⟨collection-literal-part⟩]}
⟨collection-literal-part⟩ ::= ⟨expression⟩ | ⟨expression⟩..⟨expression⟩
⟨tuple-literal⟩ ::= Tuple{[*variable* − *decl*]}

A.2 Files

The first start symbol of the grammar is ⟨file⟩. It is used to parse a separate OCL file.

⟨file⟩ ::= ⟨package-declaration⟩⟨file⟩
| ⟨context-declaration⟩⟨file⟩ | ϵ
⟨package-declaration⟩ ::= package⟨path-name⟩{⟨constraint-declaration⟩}
endpackage
⟨constraint-declaration⟩ ::= {⟨context-declaration⟩⟨constraint⟩}
⟨context-declaration⟩ ::= context⟨classifier-context⟩
| context⟨operation-context⟩
⟨classifier-context⟩ ::= ⟨path-name⟩ | ⟨identifier⟩:⟨path-name⟩
⟨operation-context⟩ ::= ⟨path-name⟩([⟨formal-parameter⟩])]
| ⟨path-name⟩([⟨formal-parameter⟩]]):⟨type⟩
⟨constraint⟩ ::= ⟨stereo-type⟩[⟨identifier⟩]:[⟨expression⟩]
| def[⟨identifier⟩]:{⟨let-expression⟩}
⟨stereo-type⟩ ::= inv | post | pre

A.3 Expressions

The second start symbol of the grammar is ⟨expression⟩. It is used to parse constraint associated to model elements in XMI files.

⟨expression⟩ ::= ⟨logical-implies-expression⟩ | ⟨let-in-expression⟩
⟨let-expression⟩ ::= let⟨identifier⟩[([⟨formal-parameter⟩]])][:⟨type⟩]
= ⟨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⟩ ::= `<` | `<=` | `>` | `>=` | `<>` | `=`

⟨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⟩

⟨simple-property-call⟩ ::= ⟨operation-name⟩[`@pre`][⟨qualifiers⟩]
[⟨poperty-call-params⟩]

⟨postfix-expression⟩ ::= ⟨primary-expression⟩`.`⟨property-call⟩
| ⟨primary-expression⟩`->`⟨property-call⟩
| ⟨primary-expression⟩`^`⟨message-call⟩
| ⟨primary-expression⟩`^^`⟨message-call⟩

⟨property-call⟩ ::= ⟨operation-name⟩[`@pre`][⟨qualifiers⟩]
[⟨poperty-call-params⟩]

⟨property-call-params⟩ ::= `(`[⟨declarator⟩]{⟨expression⟩}`)`

⟨declarator⟩ ::= {⟨identifier⟩[`:`⟨type⟩]}[`;`⟨identifier⟩[`:`⟨type⟩]
= ⟨expression⟩]|

⟨message-call⟩ ::= ⟨path-name⟩`(`{⟨message-call-argument⟩}`)`

⟨message-call-argument⟩ ::= `?`[`:`⟨type⟩] | ⟨expression⟩

⟨qualifiers⟩ ::= `[`{⟨expression⟩}`]`

⟨operation-name⟩ ::= ⟨identifier⟩ | `<` | `<=` | `>` | `>=` | `<>` | `=` | `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⟩`}}`

⟨variable-declaration⟩ ::= ⟨variable-declaration-no-init⟩[= ⟨expression⟩]

⟨variable-declaration-no-init⟩ ::= ⟨identifier⟩[`:`⟨type⟩]

Appendix B

Semantics of OCL in PVS

We summarise the formal semantics of OCL constraints in PVS. We assume extensive knowledge of PVS. If a type of a function is not defined in this appendix, it is either defined in the prelude file of the PVS 3.3 release candidate or in the PVS library of NASA Langley. All type-consistency constraints generated by the theory described here have been proved.

Recall, that in this thesis we decided to work with a *shallow embedding*. Neither the syntax nor the semantics is formalised in PVS. OCL constraints are translated into PVS constraints which have the same semantics (see Chapter 4).

The mapping from types in OCL to types in PVS is shown in Table B.1. Considering this mapping, the OCL standard library is formalised by the following PVS theory. Observe, that the type OclInvalid is not represented in the PVS theory. An expression is of this type if and only if the associated type consistency constraints do not hold. Expressions, which cannot be typed in PVS, result in inconsistent theories.

```
OCL[Classes: TYPE+, <=: (partial_order?[Classes]), Values: TYPE+
    Attributes: TYPE, References: TYPE, Locations: TYPE]: THEORY
 BEGIN
```

As displayed in the preceding fragment of PVS, the theory is parameterised of the

OCL Type	PVS Type
Boolean	bool
Integer	int
Real	real
Collection(T)	N/A
Set(T)	finite_set[T]
Sequence(T)	finite_sequence[T]
Bag(T)	finite_bag(T)
OclAny	OclAny
OclVoid	OclVoid

Table B.1: Mapping OCL types to PVS types

names of the classes occurring in the model, a partial order on these classes representing the inheritance relation, a type used to interpret attributes, the names of the attributes, the names of the association-end names (here called *references*), and the locations of the different state machines.

The operations *not*, *and*, *or*, *implies*, *iff*, and *xor* of the OCL type *Boolean* are identified with the respective functions in the PVS.

The operations +, −, ∗, /, <, >, <=, >=, *abs*, *floor*, *ceil*, *min*, and *max* of the OCL type *Real* are identified with the respective operations defined in the PVS prelude. The operation *round* is defined by expanding the definition in OCL (see Section 2.3.4), i.e.:

```
round(x: real): int = floor(x + 1/2)
```

The type Integer of OCL is identified with the type *int* in PVS. The operations +, −, ∗, /, <, >, <=, >=, *abs*, *floor*, *ceil*, *min*, and *max* are identified with the respective operations defined in the PVS prelude. The operation *mod* is identified with the function `rem` and he operation *div* is identified with `ndiv` in PVS.

The OCL types String and Unlimited Integer are not considered in our work. However, strings can be formalised using the PVS prelude. Formalising Unlimited Integer, which is a type with only one value, is in principle possible with the ordinal ω. We have doubts whether such a formalisation results in an adequate theory.

After having formalised the primitive types we define the type OclAny, i.e., the type of all objects. Observe, that we do not consider the elementary types to be subtypes of OclAny, because this would entail a rewrite of the prelude library. PVS does not allow to add new super-types to existing types. For convenience, we equate OclAny with nat.

```
OclAny: TYPE+ = nat CONTAINING 0

null: OclAny = 0

OclAnyNotNull = {obj: OclAny | obj /= null}
```

Next, we define the state of an object and the state of the system.

```
ObjectState: TYPE = [#
  class: Classes,
  location: Locations,
  aval: [Attributes -> Values],
  rval: [References -> OclAny] #]

State: TYPE = [OclAnyNotNull -> ObjectState]
```

The operations = and <> are identified with = and /= in PVS. The operations *oclIsInvalid()* and *oclIsUndefined* are not represented in PVS. In PVS they would hold if a type consistency constraint is unprovable. The operation *oclAsType* is not represented

in PVS, because the more powerful type system of PVS makes retyping unnecessary. Next, the operations *oclIsTypeOf*, *oclIsKindOf*, and *oclInState* are defined by:

```
  is_type_of(self: OclAny)(T: Type)(state: State) =
    state(self)`type = T

  is_kind_of(self: OclAny)(T: Type)(state: State) =
    state(self)`type <= T

  is_type_of_kind_of: LEMMA
    OclAny_isTypeOf(self)(T)(state) IMPLIES
      OclAny_isKindOf(self)(T)(state)

  in_state(self: OclAny)(l: Location)(state: State) =
    state(self)`location = l
```

Next we define the type OclVoid.

```
  OclVoid: TYPE FROM OclAny = {obj : OclAny | false}
```

Because OclVoid is defined as the empty type there are no operations defined for this type.

The type Collection in OCL is abstract and is not represented in PVS. If the type checker cannot determine what the concrete class of the collection is, each call to a property of the collection is translated into a case distinction in PVS. Generating the case distinction is necessary, because the type checker raises an error if it cannot resolve the type of the collection.

As displayed in Table B.1, the concrete collection types (we do not consider OrderedSet here) are identified with the corresponding finite collections in PVS. We show the definition of the conversion functions between the different collections, which are defined in separate theory (this is done to take advantage of parametric polymorphism in PVS).

```
Injections[T: TYPE]: THEORY
 BEGIN

  IMPORTING bags@top_bags

  as_set(s: finite_set[T]): finite_set[T] = s

  as_set(s: finite_sequence[T]): finite_set[T] =
      IF s`length = 0 THEN emptyset
      ELSE {e: T | EXISTS (i: below[s`length]): e = s`seq(i)}
      ENDIF
```

```
as_set(s: finite_bag[T]): finite_set[T] = bag_to_set(s)

set_to_sequence(s: finite_set[T]):
  RECURSIVE finite_sequence[T] =
    IF empty?(s) THEN empty_seq
    ELSE (# length := 1,
            seq := LAMBDA (i: below[1]): choose(s) #)
       o set_to_sequence(rest(s))
    ENDIF
   MEASURE card(s)

as_sequence(s: finite_set[T]):
    {f: finite_sequence[T] |
         IF empty?(s) THEN f = empty_seq
         ELSE EXISTS (g: [below[card(s)] -> (s)]):
                f = (# length := card(s), seq := g #) ENDIF}
```

The nondeterminism involved in converting sets to sequences is expressed by defining `as_sequence` as an uninterpreted constant and axiomatising the properties of the function. A similar declaration is used for bags below. The constructive definition of `as_sequence` is given by `set_to_sequence`. This definition is mainly used to prove the existence of a constant `as_sequence(s)` for any *s*.

```
as_sequence(s: finite_sequence[T]): finite_sequence[T] = s

as_sequence(s: finite_bag[T]):
      {f: finite_sequence[T] |
              IF empty?(s) THEN f = empty_seq
              ELSE
                EXISTS (g: [below[card(s)] ->
                                {e: T | member(e, s)}]):
                 (FORALL (e: T):
                    card({i: below[card(s)] | g(i) = e}) =
                     count(e, s))
                 AND f = (# length := card(s), seq := g #)
              ENDIF}

as_bag(s: finite_set[T]): finite_bag[T] =
    LAMBDA (e: T): IF member(e, s) THEN 1 ELSE 0 ENDIF

as_bag(s: finite_sequence[T]): finite_bag[T] =
```

```
      LAMBDA (e: T): card({i: below[s`length] | s`seq(i) = e})

  as_bag(s: finite_bag[T]): finite_bag[T] = s
 END Injections
```

Using these functions we define the meaning of `flatten` in PVS. There are nine variants, depending on the collection to be flattened *and* the type of the collection contained in it. We show only three of these functions as an example, because they all follow the same pattern.

```
Flatten[T: TYPE]: THEORY
 BEGIN

  IMPORTING Injections[T]

  IMPORTING bags@top_bags

  flatten(s: finite_set[finite_set[T]]):
   RECURSIVE finite_set[T] =
     IF empty?(s) THEN emptyset
     ELSE union(choose(s), flatten(rest(s))) ENDIF
      MEASURE card(s)

  flatten(s: finite_set[finite_sequence[T]]):
   RECURSIVE finite_set[T] =
    IF empty?(s) THEN emptyset
    ELSE union(as_set(choose(s)), flatten(rest(s))) ENDIF
     MEASURE card(s)

  flatten(s: finite_bag[finite_bag[T]]):
   RECURSIVE finite_bag[T] =
    IF nonempty_bag?(s) THEN plus(choose(s), flatten(rest(s)))
    ELSE emptybag ENDIF
   MEASURE card(s)
 END Flatten
```

Finally, we recall the definition of *iterate-expressions*. `finite_sequence` can be abbreviated as `finseq`.

```
Iterate[T: TYPE, S: TYPE]: THEORY
 BEGIN

  IMPORTING bags@top_bags
```

```
  iterate(s: finite_set[T], a: S, f: [T, S -> S]): RECURSIVE S =
    IF empty?(s) THEN a
    ELSE iterate(rest(s), f(choose(s), a), f) ENDIF
     MEASURE card(s)

  iter(s: finseq[T], a: S, f: [T, S -> S])(i: upto[s`length]):
   RECURSIVE S =
    IF i = s`length THEN a
    ELSE iter(s, f(s`seq(i), a), f)(i + 1) ENDIF
     MEASURE s`length - i

  iterate(s: finseq[T], a: S, f: [T, S -> S]): S =
    iter(s, a, f)(0)

  iterate(s: finite_bag[T], a: S, f: [T, S -> S]): RECURSIVE S =
    IF nonempty_bag?(s) THEN iterate(rest(s), f(choose(s), a), f)
    ELSE a
    ENDIF
     MEASURE card(s)
END Iterate
```

For each collection type we define its own function `iterate`. The function has the collection *s* to iterate over as its argument, an initial value for the accumulator *a*, and a function to apply for each iteration step.

Using the above definitions, we can describe the mapping of the operations defined for collections in OCL to the ones in PVS. The operations are mapped according to Table B.2.

The mapping of Sequence operations is given in Table B.3.

```
  insert_at(s: finite_sequence[T],e: T, i: posnat):
   finite_sequence[T] =
    s^(0, i-1) o
      (# length := 1, seq := LAMBDA (i: below[1]): e #) o
        s^(i, s`length)

  excluding(s: finite_sequence[T], e: T):
   RECURSIVE finite_sequence[T] =
    IF s`length = 0 THEN empty_seq
    ELSE IF s`seq(s`length - 1) = e
           THEN excluding(s ^ (0, s`length - 1), e)
         ELSE excluding(s ^ (0, s`length - 1), e) o
```

OCL Expression	Translation to PVS
$s \rightarrow size()$	`card(s)`
$s \rightarrow count(e)$	`card({x: (s) | x = s})`
$s \rightarrow includes(e)$	`member(e, s)`
$s \rightarrow excludes(e)$	`NOT member(e, s)`
$s \rightarrow includesAll(t)$	`subset(t, s)`
$s \rightarrow excludesAll(t)$	`disjoint?(s, t)`
$s \rightarrow isEmpty()$	`empty?(s)`
$s \rightarrow notEmpty()$	`NOT empty?(s)`
$s = t$	`s = t`
$s <> t$	`s /= t`
$s \rightarrow including(e)$	`union(s,{e})`
$s \rightarrow union(t)$	`union(s,t)`
$s \rightarrow intersection(t)$	`intersection(s,t)`
$s - t$	`difference(s,t)`
$s \rightarrow flatten()$	`flatten(s)` if s is a set of collections, s otherwise.
$s \rightarrow asSet()$	s
$s \rightarrow asSequence()$	`as_sequence(s)`
$s \rightarrow asBag()$	`as_bag(s)`

Table B.2: Representing Set operations in PVS

```
            (# length := 1, seq := LAMBDA (x: below[1]): e #)
      ENDIF
  ENDIF
   MEASURE s`length
```

The mapping of Bag operations is displayed in Table B.4. We use the definition of finite bags provided in the PVS library of NASA. Differences on bags in PVS is defined as:

```
difference(b,c : bag[T]): bag[T] =
  (LAMBDA (t: T): IF b(t) > c(t) THEN b(t) - c(t) ELSE 0 ENDIF)
```

This concludes the description of the semantics of OCL in PVS. Refer to Chapter 4 of how OCL expressions are embedded into PVS. Especially, Definition 4.2 defines the translation of OCL expressions into PVS expressions.

```
END OCL
```

OCL Expression	Translation to PVS
$s \rightarrow size()$	`s‘length`
$s \rightarrow count(e)$	`card({i:below[s‘length] | e = s(i)})`
$s \rightarrow includes(e)$	`EXISTS (i: below[s‘length]): e = s(i)`
$s \rightarrow excludes(e)$	`NOT EXISTS (i: below[s‘length]): e = s(i)`
$s \rightarrow includesAll(t)$	`subset?(as_set(t), as_set(s))`
$s \rightarrow excludesAll(t)$	`disjoint?(as_set(s), as_set(t))`
$s \rightarrow isEmpty()$	`s‘length = 0`
$s \rightarrow notEmpty()$	`s‘length /= 0`
$s \rightarrow at(i)$	`s(i)`
$s = t$	`s = t`
$s <> t$	`s /= t`
$s \rightarrow append(e)$	`s o (# length := 1, seq := lambda x: e #)`
$s \rightarrow prepend(e)$	`(# length := 1, seq := lambda x: e #) o s`
$s \rightarrow union(t)$	`s o t`
$s \rightarrow excluding(e)$	`excluding(s,e)`
$s \rightarrow flatten()$	`flatten(s)` if s is a sequence of collections, s otherwise.
$s \rightarrow subSequence(l, u)$	`s^(l, u)`
$s \rightarrow asSet()$	`as_set(s)`
$s \rightarrow asSequence()$	`s`
$s \rightarrow asBag()$	`as_bag(s)`

Table B.3: Representing Sequence operations in PVS

OCL Expression	Translation to PVS
$s \rightarrow size()$	`card(s)`
$s \rightarrow count(e)$	`count(s, e)`
$s \rightarrow includes(e)$	`member(s, e)`
$s \rightarrow excludes(e)$	`NOT member(s, e)`
$s \rightarrow includesAll(t)$	`FORALL e: t(e) >= s(e)`
$s \rightarrow excludesAll(t)$	`FORALL e: t(e) > 0 IMPLIES s(e) = 0`
$s \rightarrow isEmpty()$	`empty?(s)`
$s \rightarrow notEmpty()$	`NOT empty?(s)`
$s = t$	`s = t`
$s <> t$	`s /= t`
$s \rightarrow union(t)$	`plus(s, t)`
$s \rightarrow intersection(t)$	`intersection(s, t)`
$s - t$	`difference(s,t)`
$s \rightarrow excluding(e)$	`purge(s,e)`
$s \rightarrow flatten()$	`flatten(s)` if s is a bag of collections, s otherwise.
$s \rightarrow asSet()$	`as_set(s)`
$s \rightarrow asSequence()$	`as_sequence(s)`
$s \rightarrow asBag()$	`s`

Table B.4: Representing Bags operations in PVS

Bibliography

[1] Erika Ábrahám. *An Assertional Proof System for Multithreaded Java: Theory and Tool Support*. PhD thesis, Universiteit Leiden, 2005.

[2] Erika Ábrahám, Marcello M. Bonsangue, Frank S. de Boer, and Martin Steffen. Object connectivity and full abstraction for a concurrent calculus of classes. In *Proceedings of the First International Colloquium on Theoretical Aspects of Computing ICTAC 2004*, number 3704 in Lecture Notes in Computer Science, pages 38–52. Springer-Verlag, 2004.

[3] Erika Ábrahám, Frank S. de Boer, Willem-Paul de Roever, and Martin Steffen. A compositional operational semantics for Java_{MT}. In Dershowitz [48], pages 290–303.

[4] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers. Principles, Techniques, and Tools*. Addison Wesley Publishing Company, 1986.

[5] Pierre America and Frank S. de Boer. Reasoning about dynamically evolving process structures. *Formal Aspects of Computing*, 6(3):269–316, 1994.

[6] Demissie Bediye Aredo. *Formal Development of Open Distributed Systems: Integration of UML and PVS*. PhD thesis, Faculty of Mathematics and Natural Sciences, University of Oslo, 2005.

[7] Thomas Baar. *Über die Semantikbeschreibung OCL-artiger Sprachen*. PhD thesis, Fakultät für Informatik, Universität Karlsruhe, 2003. Logos Verlag, Berlin.

[8] John Warner Backus. The syntax and semantics of the proposed international algebraic language of the Zuerich acm-gramm conference. In *ICIP Paris*, June 1959.

[9] Hubert Baumeister, Rolf Hennicker, Alexander Knapp, and Martin Wirsing. OCL component invariants. In N. Chaki, editor, *Proceedings of the 8th Monterey Workshop "Engineering Automation for Software Intensive System Integration"*, pages 208–215, Monteray, California, 2001. U.S. Naval Postgraduate School.

[10] Marcello M. Bonsangue and Joost N. Kok. Infinite intersection types. *Information and Computation*, 186(2):285–318, 2003.

[11] Grady Booch. *Object-Oriented Analysis and Design with Applications*. Benjamin Cummings, 2nd edition edition, 1993.

[12] Grady Booch. Growing the UML. *Software and Systems Modeling*, 1(2):157–160, December 2002.

[13] Grady Booch, James Rumbaugh, and Ivar Jacobson. *Unified Modelling Language User Guide*. Addison Wesley Longman, 1998.

[14] Egon Börger, Alessandra Cavarra, and Elvinia Riccobene. Modeling the dynamics of UML state machines. In Yuri Gurevich, Philipp W. Kutter, Martin Odersky, and Lothar Thiele, editors, *Abstract State Machines, Theory and Applications*, volume 1912 of *Lecture Notes in Computer Science*, pages 223–241. Springer-Verlag, 2000.

[15] Nicolas Bourbaki. *Élements de Mathématique*, volume 1. Hermann, Paris, 1954.

[16] Marius Bozga, Jean-Claude Fernandez, Lucian Ghirvu, Susanne Graf, Jean-Pierre Krimm, and Laurent Mounier. IF: A validation environment for timed asynchronous systems. In E. Allen Emerson and A. Prasad Sistla, editors, *Computer Aided Verification '00*, volume 1855 of *Lecture Notes in Computer Science*, pages 543–547. Springer-Verlag, 2000.

[17] Julian C. Bradfield, Juliana Küster Filipe, and Perdita Stevens. Enriching OCL using observational mu-calculus. In Ralf-Detlef Kutsche and Herbert Weber, editors, *5th International Conference on Fundamental Approaches to Software Engineering (FASE 2002), April 2002, Grenoble, France*, volume 2306 of *Lecture Notes in Computer Science*, pages 203–217. Springer-Verlag, 2002.

[18] Ruth Breu, Ursula Hinkel, Christoph Hofmann, Cornel Klein, Barbara Paech, Bernhard Rumpe, and Veronika Thurner. Towards a formalization of the unified modeling language. In Mehmet Aksit and Satoshi Matsuoka, editors, *Proceedings of ECOOP'97 — Object-Oriented Programming, 11th European Conference*, volume 1241 of *Lecture Notes in Computer Science*. Springer-Verlag, 1997.

[19] Achim D. Brucker and Burkhart Wolff. HOL-OCL: Experiences, consequences and design choices. In Jean-Marc Jézéquel, Heinrich Hussman, and Stephen Cook, editors, *UML 2002 - The Unified Modeling Language*, volume 2460 of *Lecture Notes in Computer Science*, pages 196–211. Springer-Verlag, 2002.

[20] Achim D. Brucker and Burkhart Wolff. A proposal for a formal OCL semantics in Isabelle/HOL. In Victor Carreño, César Muñoz, and Sofiène Tashar, editors, *15th International Conference of Theorem Proving in Higher Order Logics*, volume 2410 of *Lecture Notes in Computer Science*, pages 99–114. Springer-Verlag, 2002.

[21] Luca Cardelli and Peter Wegener. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17(4):471–522, 1985.

[22] María Victoria Cengarle and Alexander Knapp. Towards OCL/RT. In Lars-Henrik Eriksson and Peter A. Lindsay, editors, *FME 2002: Formal Methods - Getting IT Right, International Symposium of Formal Methods Europe, Copenhagen, Denmark, July 22-24, 2002, Proceedings*, volume 2391 of *Lecture Notes in Computer Science*, pages 390–409. Springer-Verlag, 2002.

[23] María Victoria Cengarle and Alexander Knapp. OCL 1.4/5 vs. 2.0 expressions: Formal semantics and expressiveness. *Software and Systems Modeling*, 3(1):9–30, 2004.

[24] Peter Pin-Shan Chen. The entity-relationship model – toward a unified view of data. *ACM Transactions on Database Systems*, 1(1):9–36, March 1976.

[25] Anthony Neil Clark. Typechecking UML static models. In Robert B. France and Bernhard Rumpe, editors, *Proceedings of UML'99: The Unified Modeling Language — Beyond the Standard, Second International Conference*, volume 1723 of *Lecture Notes in Computer Science*, pages 503–517. Springer-Verlag, 1999.

[26] Anthony Neil Clark and Jos B. Warmer, editors. *Object Modelling with the OCL*, volume 2263 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002.

[27] Lauren Clark. The Apollo 35th anniversary reception. *IEEE Control Systems Magazine*, 24(6):100–101, December 2004.

[28] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Workshop on Logics of Programs*, volume 131 of *Lecture Notes in Computer Science*, pages 52–71, Yorktown Heights, New York, May 1981. Springer-Verlag. Published in 1982.

[29] Edgar F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, 1970.

[30] Adriana Beatriz Compagnoni. Higher-order subtyping and its decidability. *Information and Computation*, 191(1):41–113, 2004.

[31] Adriana Beatriz Compagnoni and Benjamin C. Pierce. Intersection types and multiple inheritance. *Mathematical Structures in Computer Science*, 6(5):469–501, October 1996.

[32] Computer Science Research Laboratory, Babes-Bolyai University of Cluj-Napoca, Romania. OCLE 1.0, 2003. `http://lci.cs.ubbcluj.ro/ocle/`.

[33] Steve Cook and John Daniels. *Designing Object Systems: Object-Oriented Modelling with Syntropy*. Prentice Hall, 1994.

[34] Steve Cook, Anneke Kleppe, Richard Mitchell, Bernhard Rumpe, Jos B. Warmer, and Alan Wills. The Amsterdam manifesto on OCL. In Clark and Warmer [26], pages 115–149.

[35] Thierre Coquand and Gérard Huet. The calculus of constructions. *Information and Computation*, 76(2/3):95–120, February/March 1988.

[36] Patrick Cousot and Rhadia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the Fourth Annual ACM Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, January 1977.

[37] Elspeth Cusack. Refinement, conformance and inheritance. *Formal Aspects of Computing*, 3(2):129–141, June 1991.

[38] Ole-Johan Dahl. Can program proving be made practical? In Michaneh Amirchahy and Danièle Néel, editors, *Les Fondements de la Programmation*, pages 57–114. INRIA, 1977.

[39] Werner Damm, Bernhard Josko, Amir Pnueli, and Angelika Votintseva. Understanding UML: A formal semantics of concurrency and communication in real-time uml. In Frank S. de Boer, Marcello Bonsangue, Susanne Graf, and Willem-Paul de Roever, editors, *Formal Methods for Components and Objects*, volume 2852 of *Lecture Notes in Computer Science*. Springer-Verlag, 2003.

[40] Dennis Dams, Rob Gerth, and Orna Grumberg. Abstract interpretation of reactive systems. *ACM Transactions on Programming Languages and Systems*, 19(2):253–291, 1997.

[41] Luca de Alfaro and Thomas A. Henzinger. Interface automata. In *Proceedings of the Ninth Annual Symposium on Foundations of Software Engineering (FSE)*, pages 109–120. ACM Press, 2001.

[42] Jaco W. de Bakker, Willem-Paul de Roever, and Grzegorz Rozenberg, editors. *Current Trends in Concurrency*, volume 224 of *Lecture Notes in Computer Science*. Springer-Verlag, 1985.

[43] Frank S. de Boer and Cees Pierik. Computer-aided specification and verification of annotated object-oriented programs. In Jacobs and Rensink [70], pages 163–177.

[44] Willem-Paul de Roever. The quest for compositionality — a survey of assertion-based proof systems for concurrent programs, Part 1: Concurrency based on shared variables. In *Proceedings of the IFIP Working Conference 1985: The Role of Abstract Models in Computer Science*, pages 181–207. North-Holland, 1985.

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

[46] Willem-Paul de Roever and Kai Engelhardt. *Data Refinement: Model-Oriented Proof Methods and their Comparison*. Number 47 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1998.

[47] María del Mar Gallardo, Pedro Merino, and Ernesto Pimentel. Debugging UML designs with model checking. *Journal of Object Technology*, 1(2):101–117, July 2002. `http://www.jot.fm/issues/issue_2002_07/article1`.

[48] Nachum Dershowitz, editor. *Proceedings of the International Symposium on Verification – Theory and Practice – Honoring Zohar Manna's 64th Birthday (Taormina, Italy, June 2003)*, volume 2772 of *Lecture Notes in Computer Science*. Springer-Verlag, 2003.

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

[50] James Clark (ed.). *XSL Transformations (XSLT) Version 1.0*. W3C, November 1999. Available for download at `http://www.w3.org/TR/xslt`.

[51] Andy Evans, Robert France, Kevin Lano, and Bernhard Rumpe. Developing the UML as a formal modelling notation. In Jean Bézevin and Pierre-Alain Muller, editors, *The Unified Modelling Language UML'98 — Beyond the Notation*, volume 1618 of *Lecture Notes in Computer Science*, pages 297–307, Berlin, Heidelberg, New-York, June 1998. Springer-Verlag.

[52] Harald Fecher, Marcel Kyas, Frank S. de Boer, and Willem-Paul de Roever. Compositional operational semantics of an UML-kernel-model language. In Peter D. Mosses and Irek Ulidowski, editors, *Proceedings of the Second Workshop on Structural Operational Semantics (SOS 2005)*, Electronic Notes in Theoretical Computer Science. Elsevier, 2005. Accepted for publication.

[53] Harald Fecher, Jens Schönborn, Marcel Kyas, and Willem-Paul de Roever. 29 new unclarities in the semantics of UML 2.0 state machines. In Kung-Kiu Lau and Richard Banach, editors, *Formal Methods and Software Engineering (ICFEM 2005)*, volume 3785 of *Lecture Notes in Computer Science*, pages 52–65. Springer-Verlag, 2005.

[54] Stephan Flake and Wolfgang Mueller. Formal semantics of OCL messages. In Peter Schmitt, editor, *Proceedings of the Workshop OCL 2.0 – Industry standard or scientific playground?*, volume 102 of *Electronic Notes in Theoretical Computer Science*, pages 77–97. Elsevier, November 2004.

[55] Martin Fowler, Martin L. Griss, Luke Hohmann, Ian Hopper, Rebecca Joos, and William F. Opdyke. *Refactoring: Improving The Design of Existing Code*. Addison-Wesley, 1999.

[56] Adele Goldberg and David Robson. *Smalltalk-80: The Language*. Addison-Wesley, 1989.

[57] James Gosling, Bill Joy, and Guy L. Steele. *The Java Language Specification*. Addison-Wesley, 3rd edition, 2005.

[58] Martin Große-Rhode. Integrating semantics for object-oriented system models. In F. Orejas, P. G. Spirakis, and J. van Leeuwen, editors, *Proceedings of the International Colloquium on Automata, Languages and Programming (ICALP 2001)*, number 2076 in Lecture Notes in Computer Science, pages 40–60. Springer Verlag, 2001.

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

[60] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, July 1987.

[61] David Harel and Eran Gery. Executable object modeling with statecharts. *Computer*, 30(7):31–42, July 1997.

[62] Rolf Hennicker, Heinrich Hußmann, and Michel Bidoit. On the precise meaning of OCL constraints. In Clark and Warmer [26], pages 69–84.

[63] Carl Hewitt. Viewing control structures as patterns of passing messages. Technical Report 410, Massachusetts Institute of Technology, Artificial Intelligence Laboratory, December 1976.

[64] Jozef Hooman. *Specification and Compositional Verification of Real-Time Systems*, volume 558 of *Lecture Notes in Computer Science*. Springer-Verlag, 1991.

[65] Jozef Hooman. Compositional verification of real-time applications. In Willem-Paul de Roever, Hans Langmaack, and Amir Pnueli, editors, *Compositionality: The Significant Difference, Proceedings of the International Symposium COMPOS '97, Malente, Germany, September 7–12, 1997*, volume 1536 of *Lecture Notes in Computer Science*, pages 276–300. Springer-Verlag, 1998.

[66] Jozef Hooman and Willem-Paul de Roever. The quest goes on: A survey of proof systems for partial correctness of CSP. In de Bakker et al. [42], pages 343–395.

[67] Jozef Hooman and Mark van der Zwaag. A semantics of communicating reactive objects with timing. In Susanne Graf, Øystein Haugen, Ileana Ober, and Bran Selic, editors, *1st Workshop on Specification and Validation of UML Models for Real Time and Embedded Systems, SVERTS 2003*, Verimag technical report 2003/10/22. Verimag, 2003. Available online at `http://www-verimag.imag.fr/EVENTS/2003/SVERTS/`.

[68] Heinrich Hußmann, Birgit Demuth, and Frank Finger. Modular architecture for a toolset supporting ocl. *Science of Computer Programming*, 44(1):51–69, 2002.

[69] Daniel Jackson. Alloy: A lightweight object modelling notation. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 11(2):256–290, April 2002.

[70] Bart Jacobs and Arend Rensink, editors. *Formal Methods for Open Object-Based Distributed Systems V*. Kluwer Academic Publishers, 2002.

[71] Ivar Jacobson, Magnus Christerson, and Patrick Jonsson. *Object-Oriented Software Engineering — A Use Case Driven Approach*. Addison-Wesley, 1992.

[72] Einar Broch Johnsen and Olaf Owe. A compositional formalism for object viewpoints. In Jacobs and Rensink [70], pages 45–60.

[73] Chris W. Johnson. The natural history of bugs: Using formal methods to analyse software related failures in space missions. In J.S. Fitzgerald, I.J. Hayes, and A. Tarlecki, editors, *Proc. Formal Methods 2005*, volume 3582 of *Lecture Notes in Computer Science*, pages 9–25. Springer-Verlag, 2005.

[74] Stephen Johnson. Lint, a C program checker. Technical Report Computer Science Technical Report 65, Bell Laboratories, December 1977.

[75] Cliff B. Jones. *Systematic Software Development using VDM*. Prentice Hall, 1990.

[76] Bengt Jonsson. A model and proof system for asynchronous networks. In *Proceeding of the Fourth Annual ACM Symposium on Principles of Distributed Computing*, pages 49–58, Minaki, Ontario, Canada, 1985. ACM Press.

[77] Stephen Cole Kleene. *Introduction to Metamathematics*. North Holland, 1952.

[78] Anneke Kleppe and Jos B. Warmer. The semantics of the OCL action clause. In Clark and Warmer [26], pages 213–227.

[79] Alexander Knapp. *A Formal Approach to Object-Oriented Software Engineering*. PhD thesis, Ludwig-Maximilians-Universität München, 2000.

[80] Cris Kobryn. UML 3.0 and the future of modeling. *Software and Systems Modeling*, 3(1):4–8, March 2004.

[81] Marcel Kyas. A compositional proof of the sieve of Eratosthenes in PVS. Technical report, Institut für Informatik, Christian-Albrechts-Universität, Kiel, Germany, 2004. Available at `http://www.informatik.uni-kiel.de/~mky/`.

[82] Marcel Kyas. An extended type system for OCL supporting templates and transformations. In Martin Steffen and Gianluigi Zavattaro, editors, *Formal Methods for Open Object-Based Distributed Systems (FMOODS 2006)*, volume 3535 of *Lecture Notes in Computer Science*, pages 83–98. Springer-Verlag, 2005.

[83] Marcel Kyas and Frank S. de Boer. Compositional specification and verification of UML models. In Paul Pettersson and Wang Yi, editors, *Proceedings of the 16th Nordic Workshop on Programming Theory*, pages 34–35, Box 377, SE-751 05 Uppsala, Sweden, October 2004. Department of Information Technology, Uppsala University. Technical Report 2004-041.

[84] Marcel Kyas and Frank S. de Boer. On message specification in OCL. In Frank S. de Boer and Marcello Bonsangue, editors, *Proceedings of the Workshop on the Compositional Verification of UML Models (CVUML)*, volume 101 of *Electronic Notes in Theoretical Computer Science*, pages 73–93. Elsevier, November 2004.

[85] Marcel Kyas, Frank S. de Boer, and Willem-Paul de Roever. A compositional trace logic for behavioural interface specifications. *Nordic Journal of Computing*, 12(2):116–132, 2005.

[86] Marcel Kyas, Harald Fecher, Frank S. de Boer, Mark van der Zwaag, Jozef Hooman, Tamarah Arons, and Hillel Kugler. Formalizing UML models and OCL constraints in PVS. In Gerald Lüttgen, Natividad Martínez Madrid, and Michael Mendler, editors, *Proceedings of Semantic Foundations of Engineering Design Languages (SFEDL 2004)*, volume 115 of *Electronic Notes in Theoretical Computer Science*, pages 39–47. Elsevier, 2005.

[87] Marcel Kyas and Jozef Hooman. Compositional verification of the MARS case study using PVS. Technical report, Institut für Informatik, Christian-Albrechts-Universität, Kiel, Germany, 2005. Available at `http://www.informatik.uni-kiel.de/~mky/pvs/mars.html`.

[88] Marcel Kyas and Jozef Hooman. Compositional verification of timed components using PVS. In Bettina Biel, Matthias Book, and Volker Gruhn, editors, *Software Engineering 2006*, volume P-79 of *Lecture Notes in Informatics*, pages 143–154. Gesellschaft für Informatik e.V., Kollen Verlag, Bonn, 2006.

[89] Leslie Lamport. *Specifying Systems*. Addison-Wesley, 2002.

[90] Leslie Lamport and Lawrence C. Paulson. Should your specification language be typed? *ACM Transactions on Programming Languages and Systems*, 21(3):502–526, May 1999.

[91] Kim Guldstrand Larsen, Paul Pettersson, and Wang Yi. Model-checking for real-time systems. In Horst Reichel, editor, *Proceedings of Fundamentals of Computation Theory*, volume 965 of *Lecture Notes in Computer Science*, pages 62–88. Springer-Verlag, 1995.

[92] Diego Latella, Istvan Majzik, and Mieke Massink. Automatic verification of a behavioural subset of UML statechart diagrams using the SPIN model-checker. *Formal Aspects of Computing*, 11(6):637–664, 1999.

[93] Daniel Leivant. Higher order logic. In Dov M. Gabbay, Christopher John Hogger, J. A. Robinson, and Jörg H. Siekmann, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming*, volume 2 – Deduction Methodologies, pages 229–321. Oxford University Press, 1994.

[94] Barbara H. Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841, November 1994.

[95] David B. MacQueen. Should ML be object-oriented? *Formal Aspects of Computing*, 13(3–5):214–232, 2002.

[96] Satoshi Matsuoka and Akinori Yonezawa. Analysis of inheritance anomaly in object-oriented concurrent programming languages. In Gul Agha, Peter Wegner, and Akinori Yonezawa, editors, *Research Directions in Concurrent Object-Oriented Programming*, pages 107–150. MIT Press, 1993.

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

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

[99] Leonid Mikhajlov and Emil Sekerinski. A study of the fragile base class problem. In Eric Jul, editor, *ECOOP'98 - Object-Oriented Programming, 12th European Conference, Brussels, Belgium, July 20-24, 1998, Proceedings*, volume 1445 of *Lecture Notes in Computer Science*, pages 355–382. Springer-Verlag, 1998.

[100] Michael Möller, Ernst-Rüdiger Olderog, Holger Rasch, and Heike Wehrheim. Linking CSP-OZ with UML and Java: A case study. In Eerke A. Boiten, John Derrick, and Graeme Smith, editors, *Integrated Formal Methods (IFM 2004)*, volume 2999 of *Lecture Notes in Computer Science*, pages 267–286. Springer-Verlag, 2004.

[101] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL – A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002.

[102] Iulian Ober, Susanne Graf, and Ileana Ober. Validation of UML models via a mapping to communicating extended timed automata. In Susanne Graf and Laurent Mounier, editors, *Model Checking Software: 11th International SPIN Workshop*, volume 2989 of *Lecture Notes in Computer Science*, pages 127–145. Springer-Verlag, 2004.

[103] Object Management Group. *OMG XMI Metadata Interchange (XMI) Specification*, June 2000. Version 1.0. Available for download at `http://cgi.omg.org/cgi-bin/doc?formal/00-06-01`.

[104] Object Management Group. *OMG XMI Metadata Interchange (XMI) Specification*, November 2000. Version 1.1. Available for download at `http://cgi.omg.org/cgi-bin/doc?formal/00-11-02`.

[105] 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`.

[106] Object Management Group. *OMG XMI Metadata Interchange (XMI) Specification*, January 2002. Version 1.2. Available for download at `http://cgi.omg.org/cgi-bin/doc?formal/02-01-01`.

[107] 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/2002-03-02`.

[108] Object Management Group. *OMG XMI Metadata Interchange (XMI) Specification*, May 2003. Version 2.0. Available for download at `http://cgi.omg.org/cgi-bin/doc?formal/03-05-02`.

[109] Object Management Group. *OMG XMI Metadata Interchange (XMI) Specification*, May 2003. Version 1.3. Available for download at `http://cgi.omg.org/cgi-bin/doc?formal/03-05-01`.

[110] Object Management Group. *UML 2.0 Infrastructure Specification*, November 2004. `http://www.omg.org/cgi-bin/doc?ptc/2004-10-14`.

[111] Object Management Group. *UML 2.0 Superstructure Specification*, October 2004. `http://www.omg.org/cgi-bin/doc?ptc/2004-10-02`.

[112] Object Management Group. *FTF Report of the OCL 2.0 Finalization Task Force*, June 2005. Available for download at `http://www.omg.org/cgi-bin/doc?ptc/2005-06-05`.

[113] Object Management Group. *OCL 2.0 Specification*, June 2005. Available for download at `http://www.omg.org/cgi-bin/doc?ptc/2005-06-06`.

[114] Ernst-Rüdiger Olderog. Process theory: Semantics, specifications and verification. In de Bakker et al. [42], pages 442–509.

[115] Ernst-Rüdiger Olderog. *Nets, Terms and Formulas: Three Views on Concurrent Processes and their Relationship*. Number 23 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1991.

[116] Omega Consortium. Omega: Correct development of real-time embedded systems, November 2003. Web-page at `http://www-omega.imag.fr`.

[117] Omega Consortium. Medium altitude reconnaissance system. Webpage at `http://www-omega.imag.fr/cs/MARS/MARS.php`, 2005.

[118] William F. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois at Urbana-Campaign, 1992.

[119] William F. Opdyke and Ralph E. Johnson. Refactoring: An aid in designing frameworks and evolving object-oriented systems. In *Proceedings of SOOPPA '90: Symposium on Object-Oriented Programming Emphasizing Practical Applications*, September 1990.

[120] Olaf Owe and Isabelle Ryl. Reasoning control in presence of dynamic classes. In *Proceedings of the 12th Workshop in Programming Theory, October 11–13, 2000, Bergen, Norway*, 2000.

[121] Sam Owre, John M. Rushby, and Natarajan Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *Automated Deduction – CADE-11*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752. Springer-Verlag, 1992.

[122] Sam Owre, John M. Rushby, Natarajan Shankar, and Friedrich von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Transactions on Software*, 21(2):107–125, 1995.

[123] Sam Owre and Natarajan Shankar. The formal semantics of PVS. Technical Report CSL-97-2R, SRI International Computer Science Laboratory, Menlo Park CA 94025 USA, 1999. August 1997, Revised March 1999.

[124] Sam Owre, Natarajan Shankar, John M. Rushby, and David W.J. Stringer-Calvert. *PVS Language Reference version 2.4*. SRI International, Computer Science Laboratory, Menlo Park, CA, dec 2001.

[125] Benjamin C. Pierce. *Programming with Intersection Types and Bounded Polymorphism*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, 1991.

[126] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[127] Cees Pierik and Frank S. de Boer. A syntax-directed hoare logic for object-oriented programming concepts. In Elie Najm, Uwe Nestmann, and Perdita Steven, editors, *Formal Methods for Open Object-Based Distributed Systems (FMOODS 2003)*, volume 2884 of *Lecture Notes in Computer Science*, pages 67–78. Springer-Verlag, 2003.

[128] Cees Pierik and Frank S. de Boer. A proof outline logic for object-oriented programming. *Theoretical Computer Science*, 343(3):413–442, October 2005.

[129] Amir Pnueli and Tamarah Arons. TLPVS: A PVS-based LTL verification system. In Dershowitz [48], pages 598–625.

[130] Jean-Pierre Queille and Joseph Sifakis. Specification and verification of concurrent systems in CESAR. In Mariangiola Dezani-Ciancaglini and Ugo Montanari, editors, *Proceedings of the 5th International Symposium on Programming*, volume 137 of *Lecture Notes in Computer Science*, pages 337–351, Paris, April 1981. Springer-Verlag.

[131] Rational Software Corporation. *UML Summary*, March 1997.

[132] Gianna Reggio, Maura Cerioli, and Egidio Astesiano. An algebraic semantics of UML supporting its multiview approach. In D. Heylen, A. Nijholt, and G. Scollo, editors, *Proc. AMiLP 2000*, 2000. Twente Workshop on Language Technology n. 16, Enschede, University of Twente.

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

[134] Mark Richters and Martin Gogolla. A semantics for OCL pre- and postconditions. In Anthony Neil Clark and Jos B. Warmer, editors, *UML 2.0 — The Future of the UML Object Constraint Language (OCL)*, October 2000. Published at `http://www.comp.brad.ac.uk/research/OCL2000/index.html` (October 17, 2005).

[135] Mark Richters and Martin Gogolla. OCL: Syntax, semantics, and tools. In Clark and Warmer [26], pages 42–68.

[136] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen. *Object-Oriented Modeling and Design*. Prentice Hall, 1990.

[137] James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1998.

[138] John M. Rushby and David W.J. Stringer-Calvert. A less elementary tutorial for the PVS specification and verification system. Technical Report CSL-95-10, SRI International Computer Science Laboratory, 1996.

[139] Michael Schenke and Ernst-Rüdiger Olderog. Transformational design of real-time systems — part 1: From requiremens to program specification. *Acta Informatika*, 36:1–65, 1999.

[140] Jens Schönborn. Formal semantics of UML 2.0 behavioral state machines. Diploma Thesis, Christian-Albrechts-Universität zu Kiel, April 2005.

[141] Andy Schürr. A new type checking approach for OCL 2.0? In Clark and Warmer [26], pages 21–40.

[142] Bran Selic, Garth Gullekson, and Paul T. Ward. *Real-Time Object-Oriented Modeling*. John Wiley & Sons, Inc., New York, Chichester, Brisbane, Toronto, Singapore, 1994.

[143] Raymond Merrill Smullyan. *First Order Logic*. Springer-Verlag, 1968.

[144] Neelam Soundararajan. Axiomatic semantics of communicating sequential processes. *ACM TOPLAS*, 6:647–662, 1984.

[145] Martin Steffen. *Polarized Higher-Order Subtyping*. PhD thesis, Technische Fakutät, Friedrich-Alexander-Universität Erlangen-Nürnberg, 1997.

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

[147] Wolfgang Thomas. Languages, automata, and logic. In Grzegorz Rozenberg and Arto Salomaa, editors, *Handbook of Formal Languages*, volume III, pages 385–455. Springer-Verlag, 1997.

[148] Issa Traoré. An outline of PVS semantics for UML statecharts. *Journal of Universal Computer Science*, 6(11):1088–1108, November 2000. `http://www.jucs.org/jucs_6_11/an_outline_of_pvs`.

[149] Mark van der Zwaag and Jozef Hooman. A semantics of communicating reactive objects with timing. *Journal on Software Tools for Technology Transfer*, 2005. Accepted for Publication in STTT.

[150] Dániel Varró. A formal semantics of UML statecharts by model transition systems. In Andrea Corradini, Hartmut Ehrig, Hans-Jörg Kreowski, and Grzegorz Rozenberg, editors, *Graph Transformation: First International Conference, ICGT 2002, Barcelona, Spain, October 7-12, 2002. Proceedings*, volume 2505 of *Lecture Notes in Computer Science*, pages 378–392. Springer-Verlag, 2002.

[151] Michael von der Beeck. A structured operational semantics for UML-statecharts. *Software and Systems Modeling*, 1(2):130–141, December 2002.

[152] Philip L. Wadler. Theorems for free! In *Fourth International Conference on Functional Programming Languages and Computer Architecture*, pages 347–359. ACM Press, 1989.

[153] Jos B. Warmer. OCL 1.4 syntax checker, 2001. `http://www.klasse.nl/ocl`.

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

[155] Jos B. Warmer and Anneke G. Kleppe. *The Object Constraint Language: Getting your models ready for MDA*. Addison-Wesley, 2nd edition edition, 2003.

[156] Pierre Wolper. The meaning of "formal": From weak to strong formal methods. *International Journal on Software Tools for Technology Transfer*, 1(1–2):6–8, December 1997.

[157] François Yergeau, Tim Bray, Jean Paoli, C.M. Sperberg-McQueen, and Eve Maler. *Extensible Markup Language (XML) 1.0*. W3C (World Wide Web Consortium), 3rd edition edition, February 2004. Available at `http://www.w3.org/TR/2004/REC-xml-20040204/`.

[158] Job Zwiers. *Compositionality, Concurrency and Partial Correctness – Proof Theories for Networks of Processes, and Their Relationship*, volume 321 of *Lecture Notes in Computer Science*. Springer-Verlag, 1989.

Summary

Embedded real-time systems are small computer systems which are used to control an increasing number of devices in every-day life. They are embedded in, for example, DVD players, microwave ovens, antilock braking systems, and autopilots. It is important that these devices always perform their function correctly in case the life of people depends upon the software used in them. Moreover, high costs are usually involved in recalling defective devices, for example, in cars. Therefore, it is desirable that these systems are formally validated, that is, a *proof* of the correct functioning of the system is constructed. Such a proof is especially important for real-time systems, because they not only need to function correctly, but also deliver their reactions *on time*. For example, an air-bag should not only inflate when a car crashes, but it should inflate milliseconds after the impact, and not seconds.

Ever since the first embedded systems were developed, their complexity has been steadily increasing. In order to control and understand this complexity different methods are used to describe the structure, the behaviour, and the requirements of software systems. Such methods are provided by the *Unified Modelling Language* (UML) and its *Object Constraint Language* (OCL) as notations (as diagrams) for describing complex *object-oriented* software systems, where the parts of these systems during execution are called *objects*. Objects react to *messages* they exchange among each other and with their *environment*, that is, with their external world. This exchange of messages is considered to be (part of) their *behaviour*.

UML provides the schema language of *class diagrams* for describing the structure, that is, the parts of the system and which parts may communicate with each other, and the notation of *state machines* for describing the behaviour of a system or its parts. OCL is used to describes the *requirements* on the system. Requirements are the properties a system has to satisfy and describe its *correct* functioning from the point of view of these given requirements.

In order to enable the development and the formal validation of these systems we have to define a *formal semantics* for the notations of UML and OCL. This means, we assign a precise meaning to the constructs of UML and OCL. This is necessary, because at present UML notations have no precise meaning. To this end, we define an unambiguous subset of UML class diagrams and define a precise mathematical semantics for this subset in Chapter 2.

UML and OCL are *typed languages*. This means that there are so-called *typing rules* on diagrams and expressions which describe when they are well-formed and therefore

have a meaning that makes sense. In Chapter 3 we show that these rules are too inflexible for writing requirements while the system is still under development. Namely, development causes changes in the system which, according to the typing rules, unexpectedly render requirements ill-formed. As a consequence, these requirements are considered nonsensical in UML. However, in our semantics they have a well-defined meaning, which has not been changed by the development step. To overcome this problem we propose extensions of the typing rules (based on so-called intersection types, union types, and bounded operator abstraction) which also improve the integration of the OCL into the UML, and which considers more requirements as well-formed.

We use *logic* to formalise the meaning of UML diagrams and OCL expressions in order to enable their formal validation. Logic makes the use of *interactive theorem provers* possible. Theorem provers assist in constructing proofs of the correct functioning of systems. This means that a system and its requirements have to be *translated* into logic. The result of this translation should be of a form that allows one to exploit all automated reasoning facilities offered by the theorem prover in finding a proof, because otherwise the construction of proofs quickly becomes complex, burdensome, and (economically) infeasible. In Chapter 4 we describe such a translation, performed by a computer program, into the input language of the theorem prover PVS and show why the translator preserves the meaning of the system and its requirements.

In order to support the specification of systems during early stages of design, we have analysed the semantics of OCL Message Expressions in Chapter 5. Message expressions specify whether messages have been sent by objects. These have been found to be inadequate. Therefore, we propose introducing *history variables* to OCL. History variables allow not only to specify and reason about the messages sent during the invocation of an operation, but also about the history of *all* messages sent and received by an object. We also show that everything which can be expressed by message expressions can also be expressed with history variables.

We strictly separate local specifications, which are requirements on the internal state of objects (and play the role of so-called *data invariants*), from local behavioural specifications, which describe the messages sent and received by an object. At a third level, we introduce global specifications which specify how objects in a system may interact.

This formalisation leads to a compositional history-based specification formalism, for which we give a compositional proof rule in Chapter 6. A specification is called *compositional* if the function of a system can be derived from the functions of its parts and the way they are put together. The main problem to solve here is the treatment of the evolution of object structures. Object structures change because objects learn about other objects during their lifetime, which enables them to communicate with new acquaintances; especially, when objects create new objects.

Finally, in Chapter 7 we extend this history-based formalism to real-time specifications. We specify a part of a *medium altitude reconnaissance system*, which is deployed by the Royal Dutch Air-Force, and prove its correctness. This example shows that the methods described in this thesis can be applied in principle to real-world case studies.

Samenvatting

Ingebedde real-time systemen zijn (kleine) computer systemen die ertoe dienen de apparaten waarin ze ingebed zijn te helpen (be)sturen. Voorbeelden van zulke apparaten zijn DVD spelers, automatische remmen, autopiloten, mobiele telefoons en Magnetic-Resonance scanners. Zulke ingebedde systemen komen meer en meer voor en worden in hoog tempo snel complexer. Ook komt het steeds vaker voor dat mensenlevens van het correct functioneren van de door hen gestuurde apparaten afhangen. Deze ontwikkeling is niet meer te stuiten. Daarom is het belangrijk dat zulke apparaten correct functioneren. En dat hangt weer af van het correcte functioneren van de hen sturende real-time systemen.

Aangezien deze systemen alom tegenwoordig zijn, zijn er industriële standaards ontwikkeld on hun functionaliteit te beschrijven. Een veel gebruikte standaard hiervoor is de UML (voor Unified Modeling Language–de naam zegt het al) en in het bijzonder zijn deeltaal OCL (voor Object Constraint Language), die ertoe dient de bedoelde betekenis van constructies in UML nader vast te leggen.

Jammer genoeg is noch de betekenis van UML, noch die van OCL eenduidig vastgelegd. (Sommige bronnen beweren dat dit met opzet gebeurd is om tegenstrijdige industriële belangen te dienen). Het is duidelijk dat als je niet precies weet wat een bepaalde taalkonstruktie betekent, je hem ook niet met 100 % zekerheid kunt gebruiken om een apparaat te sturen waar mensenlevens van afhangen.

Om in deze situatie verandering te brengen is dit proefschrift geschreven.

Het beschrijft een formele, dat wil zeggen, in wiskundige zin exacte, semantiek voor de taalconstructies van UML en OCL, en voorziet deze talen van een zinvol typesysteem dat ertoe dient om aan te geven in welke context een UML of OCL taalkonstruktie zinvol te gebruiken is. Dit type systeem is, als onderdeel vam dit proefschrift, geïmplementeerd, zodat het voldoen aan de betreffende typerings regels elektronisch kan worden gecheckt.

Om te bewijzen dat deze semantiek eenduidig is, is hij omgezet in de specificatietaal van PVS, een elektronisch systeem dat bewijzen van wiskundige stellingen op hun correctheid checkt en dat veel gebruikt wordt om er correctheidsbewijzen van programma's elektronisch mee te controleren.

Vervolgens worden in dit proefschrift een paar karakteristieke toepassingen van UML korrekt bewezen, waarbij de gebruikte semantiek die is welke in dit proefschrift vastgelegd wordt, de architectuur van deze toepassingen in UML gegeven wordt en hun functionaliteit in OCL wordt gespecificeerd.

De eerste toepassing betreft een programma voor de Zeef van Eratosthenes, dat ertoe dient de priemgetallen te genereren. Dit ontleent zijn belang aan het feit dat de desbetreffende "zeef" zich in principe een onbegrensd aantal malen (recursief) oproepen kan. Er wordt aangegeven hoe dit probleem in PVS gecodeerd kan worden, waarna de correctheid van dit programma met behulp van PVS bewezen wordt.

De tweede toepassing is ontleend aan een programma dat gebruikt wordt door de Koninklijke Luchtmacht in hun verkenningsvliegtuigen om daar zeer nauwkeurige fotos mee te maken. Wanneer namelijk vanuit straaljagers gefotografeerd wordt, moet voor nauwkeurige fotos een compensatie-mechanisme ingebouwd worden in verband met de tijdens een opname afgelegde afstand; die moet door bewegende spiegels gecompenseerd worden. Van het centrale deel van het elektronische ingebedde real-time systeem dat de beweging van deze spiegels regelt wordt een nauwkeurige specificatie in OCL gegeven en met behulp van PVS bewezen dat de UML beschrijving van de architectuur van het desbetreffende besturingssysteem aan deze specificatie voldoet.

Daarmee wordt aangetoond dat deze semantieken en hun omzetting in PVS zich er in principe toe lenen om er industriële toepassingen, waarvan architectuur en functionaliteit in UML en OCL beschreven zijn, mee korrekt te bewijzen.

Curriculum Vitæ

January 30, 1975 Born in Pinneberg, Germany.

August 1981–July 1985 Hans-Clausen-Schule, Pinneberg.

August 1985–June 1994 Diploma qualifying for university admission (Abitur) from *Johannes-Brahms-Schule*, Pinneberg, (major fields of study: mathematics and chemistry).

July 1994–September 1995 Alternative civilian service at the nursing home Haus am Rosengarten, Pinneberg.

October 1995–November 2000 Diploma from Christian-Albrechts-Universität zu Kiel (CAU) in computer science under supervision of Yassine Lakhnech and Willem-Paul de Roever. Title of diploma thesis: "Verifikation parameterisierter Netzwerke durch Abstraktion (Verification of parameterised networks by abstraction)". Minor subject: electrical engineering.

January 1997–March 1999 Student assistant (Wissenschaftliche Hilfskraft) at CAU, Institute of Economics (Operations Research), implementing a distributed version of a resource constraint scheduling problem.

April 1999–December 2000 Student assistant at CAU, Institute of Computer Science and Applied Mathematics (Software Technology), implementing static analysers for sequential function charts.

October 2000–December 2001 Assistant professor (nebenamtlicher Dozent) lecturing on *Algorithms and data structures* at FH Nordakademie.

January 2001–today Researcher at Christian-Albrechts-Universität zu Kiel, working for the IST-project *Omega* (IST-2001-33522), DFG/NWO-project *Mobi-J* (RO-1122/9-1 and RO1122/9-2), and DFG-project *SFC-Check* (LA-1021/6-1).

Current address: Christian-Albrechts-Universität zu Kiel
Institut für Informatik und Praktische Mathematik
24098 Kiel
Germany

Titles in the IPA Dissertation Series

Titles in the IPA Dissertation Series are *not* available from Lehmanns Media. Please contact the IPA Secretariat (`http://www.win.tue.nl/ipa/`) for help on obtaining a dissertation from this list.

J.O. Blanco. *The State Operator in Process Algebra*. Faculty of Mathematics and Computing Science, TUE. 1996-01

A.M. Geerling. *Transformational Development of Data-Parallel Algorithms*. Faculty of Mathematics and Computer Science, KUN. 1996-02

P.M. Achten. *Interactive Functional Programs: Models, Methods, and Implementation*. Faculty of Mathematics and Computer Science, KUN. 1996-03

M.G.A. Verhoeven. *Parallel Local Search*. Faculty of Mathematics and Computing Science, TUE. 1996-04

M.H.G.K. Kesseler. *The Implementation of Functional Languages on Parallel Machines with Distrib. Memory*. Faculty of Mathematics and Computer Science, KUN. 1996-05

D. Alstein. *Distributed Algorithms for Hard Real-Time Systems*. Faculty of Mathematics and Computing Science, TUE. 1996-06

J.H. Hoepman. *Communication, Synchronization, and Fault-Tolerance*. Faculty of Mathematics and Computer Science, UvA. 1996-07

H. Doornbos. *Reductivity Arguments and Program Construction*. Faculty of Mathematics and Computing Science, TUE. 1996-08

D. Turi. *Functorial Operational Semantics and its Denotational Dual*. Faculty of Mathematics and Computer Science, VUA. 1996-09

A.M.G. Peeters. *Single-Rail Handshake Circuits*. Faculty of Mathematics and Computing Science, TUE. 1996-10

N.W.A. Arends. *A Systems Engineering Specification Formalism*. Faculty of Mechanical Engineering, TUE. 1996-11

P. Severi de Santiago. *Normalisation in Lambda Calculus and its Relation to Type Inference*. Faculty of Mathematics and Computing Science, TUE. 1996-12

D.R. Dams. *Abstract Interpretation and Partition Refinement for Model Checking*. Faculty of Mathematics and Computing Science, TUE. 1996-13

M.M. Bonsangue. *Topological Dualities in Semantics*. Faculty of Mathematics and Computer Science, VUA. 1996-14

B.L.E. de Fluiter. *Algorithms for Graphs of Small Treewidth*. Faculty of Mathematics and Computer Science, UU. 1997-01

W.T.M. Kars. *Process-algebraic Transformations in Context*. Faculty of Computer Science, UT. 1997-02

P.F. Hoogendijk. *A Generic Theory of Data Types*. Faculty of Mathematics and Computing Science, TUE. 1997-03

T.D.L. Laan. *The Evolution of Type Theory in Logic and Mathematics*. Faculty of Mathematics and Computing Science, TUE. 1997-04

C.J. Bloo. *Preservation of Termination for Explicit Substitution*. Faculty of Mathematics and Computing Science, TUE. 1997-05

J.J. Vereijken. *Discrete-Time Process Algebra*. Faculty of Mathematics and Computing Science, TUE. 1997-06

F.A.M. van den Beuken. *A Functional Approach to Syntax and Typing*. Faculty of Mathematics and Informatics, KUN. 1997-07

A.W. Heerink. *Ins and Outs in Refusal Testing*. Faculty of Computer Science, UT. 1998-01

G. Naumoski and W. Alberts. *A Discrete-Event Simulator for Systems Engineering*. Faculty of Mechanical Engineering, TUE. 1998-02

J. Verriet. *Scheduling with Communication for Multiprocessor Computation*. Faculty of Mathematics and Computer Science, UU. 1998-03

J.S.H. van Gageldonk. *An Asynchronous Low-Power 80C51 Microcontroller*. Faculty of Mathematics and Computing Science, TUE. 1998-04

A.A. Basten. *In Terms of Nets: System Design with Petri Nets and Process Algebra*. Faculty of Mathematics and Computing Science, TUE. 1998-05

E. Voermans. *Inductive Datatypes with Laws and Subtyping – A Relational Model*. Faculty of Mathematics and Computing Science, TUE. 1999-01

H. ter Doest. *Towards Probabilistic Unification-based Parsing*. Faculty of Computer Science, UT. 1999-02

J.P.L. Segers. *Algorithms for the Simulation of Surface Processes*. Faculty of Mathematics and Computing Science, TUE. 1999-03

C.H.M. van Kemenade. *Recombinative Evolutionary Search*. Faculty of Mathematics and Natural Sciences, UL. 1999-04

E.I. Barakova. *Learning Reliability: a Study on Indecisiveness in Sample Selection*. Faculty of Mathematics and Natural Sciences, RUG. 1999-05

M.P. Bodlaender. *Scheduler Optimization in Real-Time Distributed Databases*. Faculty of Mathematics and Computing Science, TUE. 1999-06

M.A. Reniers. *Message Sequence Chart: Syntax and Semantics*. Faculty of Mathematics and Computing Science, TUE. 1999-07

J.P. Warners. *Nonlinear approaches to satisfiability problems*. Faculty of Mathematics and Computing Science, TUE. 1999-08

J.M.T. Romijn. *Analysing Industrial Protocols with Formal Methods*. Faculty of Computer Science, UT. 1999-09

P.R. D'Argenio. *Algebras and Automata for Timed and Stochastic Systems*. Faculty of Computer Science, UT. 1999-10

G. Fábián. *A Language and Simulator for Hybrid Systems*. Faculty of Mechanical Engineering, TUE. 1999-11

J. Zwanenburg. *Object-Oriented Concepts and Proof Rules*. Faculty of Mathematics and Computing Science, TUE. 1999-12

R.S. Venema. *Aspects of an Integrated Neural Prediction System*. Faculty of Mathematics and Natural Sciences, RUG. 1999-13

J. Saraiva. *A Purely Functional Implementation of Attribute Grammars*. Faculty of Mathematics and Computer Science, UU. 1999-14

R. Schiefer. *Viper, A Visualisation Tool for Parallel Program Construction*. Faculty of Mathematics and Computing Science, TUE. 1999-15

K.M.M. de Leeuw. *Cryptology and Statecraft in the Dutch Republic*. Faculty of Mathematics and Computer Science, UvA. 2000-01

T.E.J. Vos. *UNITY in Diversity. A stratified approach to the verification of distributed algorithms*. Faculty of Mathematics and Computer Science, UU. 2000-02

W. Mallon. *Theories and Tools for the Design of Delay-Insensitive Communicating Processes*. Faculty of Mathematics and Natural Sciences, RUG. 2000-03

W.O.D. Griffioen. *Studies in Computer Aided Verification of Protocols*. Faculty of Science, KUN. 2000-04

P.H.F.M. Verhoeven. *The Design of the MathSpad Editor*. Faculty of Mathematics and Computing Science, TUE. 2000-05

J. Fey. *Design of a Fruit Juice Blending and Packaging Plant*. Faculty of Mechanical Engineering, TUE. 2000-06

M. Franssen. *Cocktail: A Tool for Deriving Correct Programs*. Faculty of Mathematics and Computing Science, TUE. 2000-07

P.A. Olivier. *A Framework for Debugging Heterogeneous Applications*. Faculty of Natural Sciences, Mathematics and Computer Science, UvA. 2000-08

E. Saaman. *Another Formal Specification Language*. Faculty of Mathematics and Natural Sciences, RUG. 2000-10

M. Jelasity. *The Shape of Evolutionary Search Discovering and Representing Search Space Structure*. Faculty of Mathematics and Natural Sciences, UL. 2001-01

R. Ahn. *Agents, Objects and Events a computational approach to knowledge, observation and communication*. Faculty of Mathematics and Computing Science, TU/e. 2001-02

M. Huisman. *Reasoning about Java programs in higher order logic using PVS and Isabelle*. Faculty of Science, KUN. 2001-03

I.M.M.J. Reymen. *Improving Design Processes through Structured Reflection*. Faculty of Mathematics and Computing Science, TU/e. 2001-04

S.C.C. Blom. *Term Graph Rewriting: syntax and semantics*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2001-05

R. van Liere. *Studies in Interactive Visualization*. Faculty of Natural Sciences, Mathematics and Computer Science, UvA. 2001-06

A.G. Engels. *Languages for Analysis and Testing of Event Sequences*. Faculty of Mathematics and Computing Science, TU/e. 2001-07

J. Hage. *Structural Aspects of Switching Classes*. Faculty of Mathematics and Natural Sciences, UL. 2001-08

M.H. Lamers. *Neural Networks for Analysis of Data in Environmental Epidemiology: A Case-study into Acute Effects of Air Pollution Episodes*. Faculty of Mathematics and Natural Sciences, UL. 2001-09

T.C. Ruys. *Towards Effective Model Checking*. Faculty of Computer Science, UT. 2001-10

D. Chkliaev. *Mechanical verification of concurrency control and recovery protocols*. Faculty of Mathematics and Computing Science, TU/e. 2001-11

M.D. Oostdijk. *Generation and presentation of formal mathematical documents*. Faculty of Mathematics and Computing Science, TU/e. 2001-12

A.T. Hofkamp. *Reactive machine control: A simulation approach using χ*. Faculty of Mechanical Engineering, TU/e. 2001-13

D. Bošnački. *Enhancing state space reduction techniques for model checking*. Faculty of Mathematics and Computing Science, TU/e. 2001-14

M.C. van Wezel. *Neural Networks for Intelligent Data Analysis: theoretical and experimental aspects*. Faculty of Mathematics and Natural Sciences, UL. 2002-01

V. Bos and J.J.T. Kleijn. *Formal Specification and Analysis of Industrial Systems*. Faculty of Mathematics and Computer Science and Faculty of Mechanical Engineering, TU/e. 2002-02

T. Kuipers. *Techniques for Understanding Legacy Software Systems*. Faculty of Natural Sciences, Mathematics and Computer Science, UvA. 2002-03

S.P. Luttik. *Choice Quantification in Process Algebra*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2002-04

R.J. Willemen. *School Timetable Construction: Algorithms and Complexity*. Faculty of Mathematics and Computer Science, TU/e. 2002-05

M.I.A. Stoelinga. *Alea Jacta Est: Verification of Probabilistic, Real-time and Parametric Systems*. Faculty of Science, Mathematics and Computer Science, KUN. 2002-06

N. van Vugt. *Models of Molecular Computing*. Faculty of Mathematics and Natural Sciences, UL. 2002-07

A. Fehnker. *Citius, Vilius, Melius: Guiding and Cost-Optimality in Model Checking of Timed and Hybrid Systems*. Faculty of Science, Mathematics and Computer Science, KUN. 2002-08

R. van Stee. *On-line Scheduling and Bin Packing*. Faculty of Mathematics and Natural Sciences, UL. 2002-09

D. Tauritz. *Adaptive Information Filtering: Concepts and Algorithms*. Faculty of Mathematics and Natural Sciences, UL. 2002-10

M.B. van der Zwaag. *Models and Logics for Process Algebra*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2002-11

J.I. den Hartog. *Probabilistic Extensions of Semantical Models*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2002-12

L. Moonen. *Exploring Software Systems*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2002-13

J.I. van Hemert. *Applying Evolutionary Computation to Constraint Satisfaction and Data Mining*. Faculty of Mathematics and Natural Sciences, UL. 2002-14

S. Andova. *Probabilistic Process Algebra*. Faculty of Mathematics and Computer Science, TU/e. 2002-15

Y.S. Usenko. *Linearization in μCRL*. Faculty of Mathematics and Computer Science, TU/e. 2002-16

J.J.D. Aerts. *Random Redundant Storage for Video on Demand*. Faculty of Mathematics and Computer Science, TU/e. 2003-01

M. de Jonge. *To Reuse or To Be Reused: Techniques for component composition and construction*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2003-02

J.M.W. Visser. *Generic Traversal over Typed Source Code Representations*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2003-03

S.M. Bohte. *Spiking Neural Networks*. Faculty of Mathematics and Natural Sciences, UL. 2003-04

T.A.C. Willemse. *Semantics and Verification in Process Algebras with Data and Timing*. Faculty of Mathematics and Computer Science, TU/e. 2003-05

S.V. Nedea. *Analysis and Simulations of Catalytic Reactions*. Faculty of Mathematics and Computer Science, TU/e. 2003-06

M.E.M. Lijding. *Real-time Scheduling of Tertiary Storage*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2003-07

H.P. Benz. *Casual Multimedia Process Annotation – CoMPAs*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2003-08

D. Distefano. *On Modelchecking the Dynamics of Object-based Software: a Foundational Approach*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2003-09

M.H. ter Beek. *Team Automata – A Formal Approach to the Modeling of Collaboration Between System Components*. Faculty of Mathematics and Natural Sciences, UL. 2003-10

D.J.P. Leijen. *The λ Abroad – A Functional Approach to Software Components*. Faculty of Mathematics and Computer Science, UU. 2003-11

W.P.A.J. Michiels. *Performance Ratios for the Differencing Method*. Faculty of Mathematics and Computer Science, TU/e. 2004-01

G.I. Jojgov. *Incomplete Proofs and Terms and Their Use in Interactive Theorem Proving*. Faculty of Mathematics and Computer Science, TU/e. 2004-02

P. Frisco. *Theory of Molecular Computing – Splicing and Membrane systems*. Faculty of Mathematics and Natural Sciences, UL. 2004-03

S. Maneth. *Models of Tree Translation*. Faculty of Mathematics and Natural Sciences, UL. 2004-04

Y. Qian. *Data Synchronization and Browsing for Home Environments*. Faculty of Mathematics and Computer Science and Faculty of Industrial Design, TU/e. 2004-05

F. Bartels. *On Generalised Coinduction and Probabilistic Specification Formats*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2004-06

L. Cruz-Filipe. *Constructive Real Analysis: a Type-Theoretical Formalization and Applications*. Faculty of Science, Mathematics and Computer Science, KUN. 2004-07

E.H. Gerding. *Autonomous Agents in Bargaining Games: An Evolutionary Investigation of Fundamentals, Strategies, and Business Applications*. Faculty of Technology Management, TU/e. 2004-08

N. Goga. *Control and Selection Techniques for the Automated Testing of Reactive Systems*. Faculty of Mathematics and Computer Science, TU/e. 2004-09

M. Niqui. *Formalising Exact Arithmetic: Representations, Algorithms and Proofs*. Faculty of Science, Mathematics and Computer Science, RU. 2004-10

A. Löh. *Exploring Generic Haskell*. Faculty of Mathematics and Computer Science, UU. 2004-11

I.C.M. Flinsenberg. *Route Planning Algorithms for Car Navigation*. Faculty of Mathematics and Computer Science, TU/e. 2004-12

R.J. Bril. *Real-time Scheduling for Media Processing Using Conditionally Guaranteed Budgets*. Faculty of Mathematics and Computer Science, TU/e. 2004-13

J. Pang. *Formal Verification of Distributed Systems*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2004-14

F. Alkemade. *Evolutionary Agent-Based Economics*. Faculty of Technology Management, TU/e. 2004-15

E.O. Dijk. *Indoor Ultrasonic Position Estimation Using a Single Base Station*. Faculty of Mathematics and Computer Science, TU/e. 2004-16

S.M. Orzan. *On Distributed Verification and Verified Distribution*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2004-17

M.M. Schrage. *Proxima - A Presentation-oriented Editor for Structured Documents*. Faculty of Mathematics and Computer Science, UU. 2004-18

E. Eskenazi and A. Fyukov. *Quantitative Prediction of Quality Attributes for Component-Based Software Architectures*. Faculty of Mathematics and Computer Science, TU/e. 2004-19

P.J.L. Cuijpers. *Hybrid Process Algebra*. Faculty of Mathematics and Computer Science, TU/e. 2004-20

N.J.M. van den Nieuwelaar. *Supervisory Machine Control by Predictive-Reactive Scheduling*. Faculty of Mechanical Engineering, TU/e. 2004-21

E. Ábrahám. *An Assertional Proof System for Multithreaded Java -Theory and Tool Support-* . Faculty of Mathematics and Natural Sciences, UL. 2005-01

R. Ruimerman. *Modeling and Remodeling in Bone Tissue*. Faculty of Biomedical Engineering, TU/e. 2005-02

C.N. Chong. *Experiments in Rights Control - Expression and Enforcement*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-03

H. Gao. *Design and Verification of Lock-free Parallel Algorithms*. Faculty of Mathematics and Computing Sciences, RUG. 2005-04

H.M.A. van Beek. *Specification and Analysis of Internet Applications*. Faculty of Mathematics and Computer Science, TU/e. 2005-05

M.T. Ionita. *Scenario-Based System Architecting - A Systematic Approach to Developing Future-Proof System Architectures*. Faculty of Mathematics and Computing Sciences, TU/e. 2005-06

G. Lenzini. *Integration of Analysis Techniques in Security and Fault-Tolerance*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-07

I. Kurtev. *Adaptability of Model Transformations*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-08

T. Wolle. *Computational Aspects of Treewidth - Lower Bounds and Network Reliability*. Faculty of Science, UU. 2005-09

O. Tveretina. *Decision Procedures for Equality Logic with Uninterpreted Functions*. Faculty of Mathematics and Computer Science, TU/e. 2005-10

A.M.L. Liekens. *Evolution of Finite Populations in Dynamic Environments*. Faculty of Biomedical Engineering, TU/e. 2005-11

J. Eggermont. *Data Mining using Genetic Programming: Classification and Symbolic Regression*. Faculty of Mathematics and Natural Sciences, UL. 2005-12

B.J. Heeren. *Top Quality Type Error Messages*. Faculty of Science, UU. 2005-13

G.F. Frehse. *Compositional Verification of Hybrid Systems using Simulation Relations*. Faculty of Science, Mathematics and Computer Science, RU. 2005-14

M.R. Mousavi. *Structuring Structural Operational Semantics*. Faculty of Mathematics and Computer Science, TU/e. 2005-15

A. Sokolova. *Coalgebraic Analysis of Probabilistic Systems*. Faculty of Mathematics and Computer Science, TU/e. 2005-16

T. Gelsema. *Effective Models for the Structure of pi-Calculus Processes with Replication*. Faculty of Mathematics and Natural Sciences, UL. 2005-17

P. Zoeteweij. *Composing Constraint Solvers*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2005-18

J.J. Vinju. *Analysis and Transformation of Source Code by Parsing and Rewriting*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2005-19

M.Valero Espada. *Modal Abstraction and Replication of Processes with Data*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2005-20

A. Dijkstra. *Stepping through Haskell*. Faculty of Science, UU. 2005-21

Y.W. Law. *Key management and link-layer security of wireless sensor networks: energy-efficient attack and defense*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-22

E. Dolstra. *The Purely Functional Software Deployment Model*. Faculty of Science, UU. 2006-01

R.J. Corin. *Analysis Models for Security Protocols*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2006-02

P.R.A. Verbaan. *The Computational Complexity of Evolving Systems*. Faculty of Science, UU. 2006-03

K.L. Man and R.R.H. Schiffelers. *Formal Specification and Analysis of Hybrid Systems*. Faculty of Mathematics and Computer Science and Faculty of Mechanical Engineering, TU/e. 2006-04

M. Kyas. *Verifying OCL Specifications of UML Models: Tool Support and Compositionality*. Faculty of Mathematics and Natural Sciences, UL. 2006-05

Stellingen
behorende bij het proefschrift

Verifying OCL Specifications of UML Models: Tool Support and Compositionality

door
Marcel Kyas

I

When specifying systems one has to be aware of the subtle differences between *null* and *undefined*: Any programmer expects that *null* = *null* is true and that *undefined* = *undefined* is nonsense.

II

OCL cannot be used to specify the behaviour of operations, because: (i) the specification may call operations defined in the model as long as they are side-effect free, (ii) these operations can be overridden, even if they are defined in the OCL standard library, and (iii) *virtual binding* is used to resolve such calls. As a consequence, the meaning of constraints in a class diagram depends on its implementation.

III

Lamport and Paulson hold the opinion that mathematicians are so intelligent that their specification languages do not need to be typed [LP99]. Specification languages like OCL demonstrate the contrary.

IV

Karl Popper's remark that "whenever a theory appears to you as the only possible one, take this as a sign that you have neither understood the theory nor the problem which it was intended to solve" [Pop72] holds especially for UML.

V

UML 2.0 state machines can be rigorously formalised in about ten pages of rewriting logic [Sch05], which expose all ambiguities and unclarities [FSKdR05] occurring in the 68 page description in UML 2.0 [Obj04].

VI

UML state machines improve drastically on most *modern* object-oriented programming languages, whose semantics is based on ALGOL-60, by basing their semantics on Hewitt's actor model [Hew76].

VII

Some of the problems of proving industrial applications correct are: (i) The given specification is almost never correct. (ii) The given application is not structurally described, i.e., by composing simpler constructs to complicated ones in a hierarchical manner, also called by stepwise hierarchical refinement.

VIII

Completeness results are only relevant if the proof of completeness shows a generally applicable method for *de facto* constructing a proof for a correct program.

IX

Old-Norse poetry like *Ynglingatal* [AM 45 fol.] has been recorded by Christians. We have to mind this fact when arguing whether there was *sacral kingship* (where a king is viewed as a mediator or executive agent of a god) in pre-Christian Scandinavia of which features have been maintained by kings of Christian medieval

Scandinavia [Bae64]. We must also not forget that *our* reception of these poems is heavily influenced by our own culture [Fro51], which is strongly affected by Christianity.

X

The main problem of designing a distributed version of a Linda-tuple-space is not that Linda is inherently inefficient, but that it is difficult to find *reasonable* fairness requirements [Der05, Hlu05].

XI

Paul Lorenzen devised game semantics *(Dialogische Logik)*, because *every* scientist, especially humanists, should be able to reason formally [KL96]. However, most non-logicians do not apprehend game semantics.

XII

If inventions can be patented that do not make causally determined use of natural matter and energy, as is the case with software, then *all* teaching concerning mental activity becomes susceptible to patent litigation.

References

[AM 45 fol.] Am 45 fol. Codex Frisianus. Arnamagnæan Collection. Copenhagen, Denmark, ca. 1300–1325.

[Bae64] Walter Baetke. Yngvi und die Ynglinger. Eine quellenkritische Untersuchung über das nordische "Sakralkönigtum". *Sitzungsberichte der Sächsischen Akademie der Wissenschaften zu Leipzig*, 109(3), 1964.

[Der05] Alexander Derenbach. Client/Server-Architektur und Servertopologien eines verteilten Linda-Tupelraum in Java. Bachelor Thesis, Christian-Albrechts-Universität zu Kiel, October 2005.

[Fro51] Erich Fromm. *The Forgotten Language: An Introduction to the Understanding of Dreams, Fairy Tales and Myths*. Rinehart and Co., 1951.

[FSKdR05] Harald Fecher, Jens Schönborn, Marcel Kyas, and Willem-Paul de Roever. 29 new unclarities in the semantics of UML 2.0 state machines. In Kung-Kiu Lau and Richard Banach, editors, *Formal Methods and Software Engineering (ICFEM 2005)*, volume 3785 of *Lecture Notes in Computer Science*, pages 52–65. Springer-Verlag, 2005.

[Hew76] Carl Hewitt. Viewing control structures as patterns of passing messages. Technical Report 410, Massachusetts Institute of Technology, Artificial Intelligence Laboratory, December 1976.

[Hlu05] Christopher Hlubek. Eine verteilte Tupelraum Implementierung in Java. Bachelor Thesis, Christian-Albrechts-Universität zu Kiel, October 2005.

[KL96] Wilhelm Kamlah and Paul Lorenzen. *Logische Propädeutik: Vorschule des vernünftigen Redens*. J.B. Metzler, Stuttgart, Weimar, 3rd edition, 1996.

[LP99] Leslie Lamport and Lawrence C. Paulson. Should your specification language be typed? *ACM Transactions on Programming Languages and Systems*, 21(3):502–526, May 1999.

[Obj04] Object Management Group. *UML 2.0 Superstructure Specification*, October 2004. `http://www.omg.org/cgi-bin/doc?ptc/2004-10-02`.

[Pop72] Karl Raymond Popper. *Objective Knowledge: An Evolutionary Approach*. Oxford University Press, 1972.

[Sch05] Jens Schönborn. Formal semantics of UML 2.0 behavioral state machines. Diploma Thesis, Christian-Albrechts-Universität zu Kiel, April 2005.