Verifying OCL specifications of UML models : tool support and compositionality

Kyas, M.

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.