PMD '01

Precise Modelling and Deduction for Object-oriented Software Development

International Workshop at IJCAR 2001

Sienna, June 2001

Bernhard Beckert
Robert France
Reiner Hähnle
Bart Jacobs
(eds.)

Table of Contents

Preface

The *International Workshop on Precise Modelling and Deduction for Object-oriented Software Development* is held on June 18, 2001 in Siena (Italy) as part of IJCAR 2001, the International Joint Conference on Automated Reasoning.

The workshop aims at closing the gap between automated deduction and one of its most important applications: formal methods in software engineering. It tries to bring together the precise modelling and the automated reasoning communities interested in object-oriented software development.

The meeting consists of an invited talk, five contributed papers, and a tool demo session. More information, including these proceedings, is available at the PMD web site at `i12www.ira.uka.de/~beckert/PMD`.

We sincerely thank those who contributed to make this workshop possible: The authors and participants, and in particular Johann Schumann for his invited talk. We also thank the organisers of IJCAR 2001, who were responsible for all local arrangements.

Sienna, Italy
June 2001

Bernhard Beckert
Robert France
Reiner Hähnle
Bart Jacobs

Automatic Synthesis of UML Designs from Requirements in an Iterative Process

Johann Schumann[1] and Jon Whittle[2]

[1] RIACS/NASA-Ames
[2] QSS/NASA Ames
email:schumann,jonathw@ptolemy.arc.nasa.gov

The Unified Modeling Language (UML) is gaining wide popularity for the design of object-oriented systems. UML [6] combines various object-oriented graphical design notations under one common framework. A major factor for the broad acceptance of UML is that it can be conveniently used in a highly iterative, Use Case (or scenario-based) process (although the process is not a part of UML). Here, the (pre-)requirements for the software are specified rather informally as Use Cases and a set of scenarios. A scenario can be seen as an individual trace of a software artifact. Besides first sketches of a class diagram to illustrate the static system breakdown, scenarios are a favorite way of communication with the customer, because scenarios describe concrete interactions between entities and are thus easy to understand. Scenarios with a high level of detail are often expressed as sequence diagrams.

Later in the design and implementation stage (elaboration and implementation phases), a design of the system's behavior is often developed as a set of statecharts. From there (and the full-fledged class diagram), actual code development is started. Current commercial UML tools support this phase by providing code generators for class diagrams and statecharts.

In practice, it can be observed that the transition from requirements to design to code is a highly iterative process. This means that initial versions of requirements have to be modified and refined to meet additional (customer) wishes and constraints. Also modifications of the code can lead to revisions in design. This iterative behavior is strongly supported by most modern processes, because it facilitates early detection of inconsistencies and bugs. Fixing a bug which is detected late in the software lifecycle can cost approximately 60-100 times more than one which is detected early [3].

However, current UML tools do not support the transition from requirements to design in a comfortable and consistent way. Often, a considerable amount of time is spent to write down the requirements in great detail. Then the requirements tend to be "forgotten" until test cases have to be set up. At this point of time, it is usually detected that those requirements are hopelessly out of date and require a major overhaul.

Our work [7] addresses these issues and tries to close the gap between requirements and design. In this talk, we present a set of algorithms which perform reasonable synthesis and transformations between different UML notations (sequence diagrams, OCL constraints, statecharts). Our overall aim with respect to reasonable synthesis is centered around the following concepts: detection of inconsistencies and ambiguities in sequence diagrams, merging of similar or duplicated behaviors from different sequence diagrams, the production of highly readable (structured) statechart, and the support for iterative refinements. More specifically, we will discuss the following transformations.

Statechart synthesis. From a set of sequence diagrams with object O (as an instance of a class C) as a participant, we automatically synthesize a statechart which reflects C's behavior given in the sequence diagrams. Because the standard semantics of sequence diagrams is very weak, almost no duplicate or similar behavior can be merged. In order to overcome this problem, we allow the designer to specify a set of OCL constraints, describing pre- and postconditions over a vector of "state-variables" for messages in the sequence diagrams. These state-variables (currently of type boolean) and the constraints are used by our algorithm to detect conflicts between a sequence diagram and the OCL constraints (the domain model) using unification and a version of the frame axiom. Furthermore, potential loops can be detected. Our state variables also form the basis for constructing the (flat) statechart. In contrast to other approaches (e.g., that used in the SCED tool [2]), the domain model allows a justified merge of sequence diagrams. Because OCL constraints need to be defined only for few (possibly important or ambiguous) messages, we believe that the additional burden for the designer is kept to a reasonable level.

Introduction of hierarchy. As soon as the design gets more complex (i.e., a statechart contains more than approx. 5 nodes), things usually get out of hand, because the design cannot be read by the designer/developer in a reasonable manner. D. Harel [1] tackled this problem by introducing hierarchy and orthogonality in his statecharts. Nodes can be grouped into supernodes, increasing readability and avoiding an explosion of states when new functionality is added.

In order to produce useful designs, our algorithm is capable of synthesizing hierarchical statecharts. Thereby, the initial flat statechart is partitioned recursively according to a given strategy, usually based upon information in the class diagram, a given ordering of the state-variables, and user preferences. Because hierarchy is transparent with respect to statechart semantics, multiple different hierarchies (or "views") can exist in the system at the same time.

Consistency of modifications. In most software projects, requirements scenarios only cover a (hopefully important) fragment of the intended system behavior. Therefore, the synthesized statechart can only be a first design sketch which needs to be generalized and modified by the designer. A hierarchical structure (see above) is an important prerequisite for such activities. However, transformations or modifications easily can invalidate the requirements. Therefore, we have developed a "backwards direction" algorithm which checks consistency of the modified statechart with the original requirements and the domain model. In case an original sequence diagram has been violated, our algorithm proposes a set of revised (added/modified/deleted messages) according to given criteria.

"Design-Debugging". Despite the well-known "fact" that every programmer always writes error-free code, debugging of a software artifact is an extremely important (and unfortunately time-consuming and costly) task. Our algorithms support debugging of UML diagrams on various levels [5]. Early checking of consistency in the requirements is one way of debugging during very early stages of the development, i.e., already before the actual design starts. Our backwards-direction algorithm facilitates finding bugs in modifications of the original design. Here, the user is not required to manually go through (lengthy) execution traces. All the user has to do is to check the proposed modifications (which are usually much smaller) of the sequence diagrams whether or not they are consistent with the intended system behavior.

A popular method for debugging is the so-called "printf-debugging". Here, the programmer instruments the code with statements which write trace information and variable values into a log-file. After the program execution, the trace in analyzed. In practice, however, annotation of larger program to detect a certain behavior is far from trivial. Usually, a lot of distant and seemingly unrelated parts of the code have to be annotated. Here, our algorithm for the introduction of hierarchy can be of great help. Combined with the automatic code generation facilities of commercial UML tools, such an instrumentation can be accomplished easily. The developer changes the hierarchy of the statechart(s) in such a way that all states which are of interest for the current debugging session are grouped together in one (or a few) superstates on the top of the hierarchy. Then, all important parts are clearly visible and can be instrumented easily (e.g., by adding specific debugging actions). The change of the hierarchy can be initiated by giving additional constraints over the state variables.

Our entire set of algorithms is based upon a logic-based semantics of the different UML notations. We are currently only using a subset of the sequence diagram and statechart notation, for which there is a straightforward, undisputed semantics. In future, we will work on the incorporation of additional elements of the statechart notation and extensions of sequence diagrams (see [8] for details) into our framework.

We have developed a prototype of these algorithms in Java. Integration into a UML tool (using XMI) is currently in progress. We have tried out our algorithm with various small examples, like the ATM machine and a cruise-control system. Future work includes NASA-internal case studies on space shuttle software and software for advanced air traffic control.

However, there is much work still to be done. Our overall goal is to have an integrated UML support tool which is concise and accurate, but hides the underlying formal techniques (unification, constraint solving, tree searches) as much as possible. By integration of the algorithms into commercial UML tools we aim at "invisible formal methods" as proposed by J. Rushby [4]. The incorporation of additional domain information in the form of OCL constraints allows concise consistency checks and justified merging of sequence diagrams with minimal overhead for the software designer and developer. It is thus expected that such tools will increase productivity and quality of object oriented software systems.

References

1. D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.
2. T. Männistö, T. Systä, and J. Tuomi. SCED report and user manual. Report A-1994-5, Dept of Computer Science, University of Tampere, 1994.
3. R. Pressman. *Software Engineering - a Practitioner's Approach*. McGraw-Hill, 1997.
4. J. Rushby. Disappearing formal methods. In *Proceedings of HASE: Fifth IEEE International Symposium on High Assurance Systems Engineering*, 2000. invited paper.
5. J. Schumann. Automatic debugging support for UML designs. In M. Ducasse, editor, *Proceedings of the Fourth International Workshop on Automated Debugging*, 2000. http://xxx.lanl.gov/abs/cs.SE/0011017.
6. Unified Modeling Language Specification, Version 1.3, 1999. Available from Rational Software Corporation, Cupertino, CA.
7. J. Whittle and J. Schumann. Generating Statechart Designs From Scenarios. In *Proceedings of International Conference on Software Engineeering (ICSE 2000)*, pages 314–323, 2000.
8. J. Whittle and J. Schumann. Generating Statechart Designs From Scenarios. *TOSEM*, 2001. submitted.

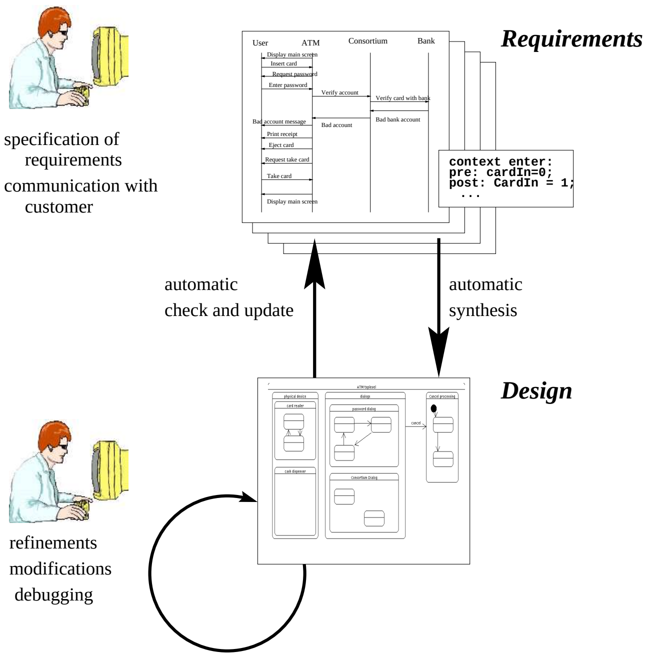

Fig. 1. Automatic synthesis of statecharts in a highly iterative software process

Handling Java's Abrupt Termination in a Sequent Calculus for Dynamic Logic

Bernhard Beckert and Bettina Sasse

University of Karlsruhe
Institute for Logic, Complexity and Deduction Systems
D-76128 Karlsruhe, Germany
beckert@ira.uka.de, sasse@ira.uka.de

Abstract. In JAVA, the execution of a statement can terminate abruptly (besides terminating normally and terminating not at all). Abrupt termination either leads to a redirection of the control flow after which the program execution resumes (for example if an exception is caught), or the whole program terminates abruptly (if an exception is not caught). Within the KeY project, a Dynamic Logic for Java Card has been developed, as well as a sequent calculus for that logic, which can be used to verify JAVA CARD programs. In this paper, we describe how abrupt termination is handled in that calculus. The ideas behind the rules we present can easily be adapted to other program logics (in particular Hoare logic) for JAVA.

1 Introduction

In JAVA, the execution of a statement can terminate *abruptly* (besides terminating normally and terminating not at all). Possible reasons for an abrupt termination are for instance (a) that an exception has been thrown, (b) that a loop or a single loop iteration is terminated with the `break` resp. the `continue` statement, and (c) that the execution of a method is terminated with the `return` statement. Abrupt termination of a statement either leads to a redirection of the control flow after which the program execution resumes (for example if an exception is caught), or the whole program terminates abruptly (if an exception is not caught).

In [2] a Dynamic Logic for JAVA CARD (JAVA CARD DL) has been presented, as well as the basic rules of a sequent calculus for JAVA CARD DL that can be used to verify JAVA CARD programs. In this paper, we give a detailed description of how abrupt termination is handled in that calculus. The basic principles of the rules we present can easily be adapted to other program logics (in particular Hoare logic) for JAVA.

The basic idea of our approach, which helps to keep the calculus's rules simple, is to give an *abruptly* terminating statement the same semantics as that of a *non-terminating* statement. As usual in Dynamic Logics, the semantics of a program is a partial functions between states. Neither the fact that an abrupt termination has occurred nor the reason for the abrupt termination are made part of the states. Thus, to define the semantics of DL formulas, we do not need to provide additional constructs for handling abrupt termination. Nevertheless, our calculus can handle programs that make use of abrupt termination to redirect control flow during execution.

We work according to the principle that the program states should not include information about control flow: they do not contain a program counter, nor the value of the condition in an `if-else` statement that has just been evaluated, *nor the reason for the termination of a statement.*

A different approach is used in [3], where the semantics of a program is not a function between states but from states to pairs consisting of a state and a reason for termination, making the reason for completion effectively part of the final state of a statement. Other related work includes [6] and [8], where program logics for (subsets of) Java are described.

The structure of this paper is as follows: In Section 2, we shortly describe the background and motivation of our work. Syntax and semantics of Java Card DL are introduced in Section 3; for details, the reader is referred to [2]. The rules for handling abrupt termination are given in Section 4. In Section 5, we present an example for the application of these rules.

2 Background

The work reported here has been carried out as part of the KeY project [1]. The goal of KeY is to enhance a commercial CASE tool with functionality for formal specification and deductive verification and, thus, to integrate formal methods into real-world software development processes. Accordingly, the design principles for the software verification component of the KeY system are:

– The programs that are verified should be written in a "real" object-oriented programming language (we decided to use Java Card).
– The logical formalism should be as easy as possible to use for software developers (who do not have years of training in formal methods).

Since Java Card is a "real" object-oriented language, it has features which are difficult to handle in a software verification system, such as dynamic binding, aliasing, object initialisation, and—the topic of this paper—abrupt termination. On the other hand, Java Card lacks some crucial complications of the full Java language such as threads and dynamic loading of classes. Moreover, Java smart cards are an extremely suitable target for software verification, as the applications are typically security-critical but rather small.

We use an instance of Dynamic Logic (DL) [5]—which can be seen as an extension of Hoare logic—as the logical basis of the KeY system's software verification component, because deduction in DL is based on symbolic program execution and simple program transformations and is close to a programmer's understanding of Java Card. Also, DL has successfully been applied in practice to verify software systems of considerable size. It is used in the software verification systems KIV [7] and VSE [4] (for a programming language that is not object-oriented).

3 Dynamic Logic for Java Card

3.1 Overview

Dynamic Logic can be seen as a modal predicate logic with a modality $\langle p \rangle$ for every program p (we allow p to be any sequence of legal Java Card statements); $\langle p \rangle$ refers to the successor worlds (called states in the DL framework) that are reachable by running the program p. In standard DL there can be several of these states (worlds) because the programs can be non-deterministic; but here, since Java Card programs are deterministic, there is exactly one such world (if p terminates) or there is no such world (if p does not terminate). The formula $\langle p \rangle \phi$ expresses that the program p terminates in a state in which ϕ holds. A formula $\phi \to \langle p \rangle \psi$ is valid if for every state s satisfying the pre-condition ϕ, a run of the program p starting in s terminates, and in the terminating state the post-condition ψ holds.

Thus, the formula $\phi \to \langle p \rangle \psi$ is similar to the Hoare triple $\{\phi\} p \{\psi\}$. But in contrast to Hoare logic, the set of formulas of DL is closed under the usual logical operators: In Hoare logic, the formulas ϕ and ψ are pure first-order formulas. DL allows to involve programs in the descriptions ϕ resp. ψ of states. For example, using a program, it is easy to specify that a data structure is not cyclic, which is impossible in pure first-order logic. Because all JAVA constructs are available in DL for the description of states (including `while` loops and recursion) it is not necessary to define an abstract data type *state* and to represent states as terms of that type; instead DL formulas can be used to give a (partial) description of states, which is a more flexible technique and allows to concentrate on the relevant properties of a state.

3.2 Syntax of Java Card DL

As said above, a dynamic logic is constructed by extending some non-dynamic logic with modal operators of the form $\langle p \rangle$. The non-dynamic base logic of our DL is a typed first-order predicate logic. We do not describe in detail what the types of our logic are (basically they are identical with the JAVA types) nor how exactly terms and formulas are built, as this is not relevant for the handling of abrupt termination. The definitions can be found in [2]. Note, that terms (which we often call "logical terms" in the following) are different from JAVA expressions; they never have side effects.

In order to reduce the complexity of the programs occurring in DL formulas, we introduce the notion of a *program context*. The context can consist of any legal JAVA CARD program, i.e., it is a sequence of class and interface definitions. Syntax and semantics of DL formulas are then defined with respect to a given context; and the programs in DL formulas are assumed not to contain class definitions.

A context must not contain any constructs that lead to a compile-time error or that are not available in JAVA CARD.[1]

The programs in DL formulas are basically executable JAVA CARD code; as said above, they must not contain class definitions but can only use classes defined in the program context. We introduced two additional constructs that are not available in plain JAVA CARD but are necessary for certain rule applications: Programs can contain a special construct for method invocation (see below), and they can contain logical terms. These extensions are not used in the input formulas, they occur only within proofs, i.e., we prove properties of pure JAVA CARD programs.

Example 1. The statement `i=0;` may be used as a program in a DL formula although `i` is not declared as a local variable.

The statement `break l;` is `not` a legal program because such a statement is only allowed to occur inside a block labelled with `l`. Accordingly, `l:{break l;}` is a legal program and can be used in a DL formula.

[1] An additional restriction is that a program context must not contain *inner classes* (this restriction is "harmless" because inner classes can be removed with a structure-preserving program transformation and are rarely used in JAVA CARD anyway).

The purpose of our first extension is the handling of method calls. Methods are invoked by syntactically replacing the call by the method's implementation. To handle the `return` statement in the right way, it is necessary (a) to record the object field or variable *x* that the result is to be assigned to, (b) to record the old value *old* of `this`, and (c) to mark the boundaries of the implementation *prog* when it is substituted for the method call. For that purpose, we allow statements of the form `call(`*old*`,`*x*`){`*prog*`}` to occur in DL programs.

The second extension is to integrate logical terms in programs contained in DL formulas (not in the program context). This is necessary to be able to replace Java expressions with possible side effects by a logical term of the same type. However, since the value of logical terms cannot and must not be changed by a program, a logical term can only be used in positions where a `final` local variable could be used according to the Java language specification (the value of local variables that are declared `final` cannot be changed either). In particular, logical terms cannot be used as the left hand side of an assignment.

3.3 Semantics of Java Card DL

The semantics of a program *p* is a state transition, i.e., it assigns to each state s the set of all states that can be reached by running *p* starting in s. Since Java Card is deterministic, that set either contains exactly one state (if *p* terminates normally) or is empty (if *p* does not terminate or terminates abruptly). The set of states of a model must be closed under the reachability relation for all programs *p*, i.e., all reachable states must exist in a model (other models are not considered).

The semantics of a logical term t occurring in a program is the same as that of a Java expression whose evaluation is free of side-effects and gives the same value as t.

For formulas ϕ that do not contain programs, the notion of ϕ being satisfied by a state is defined as usual in first-order logic. A formula $\langle p \rangle\phi$ is satisfied by a state s if the program p, when started in s, terminates normally in a state s' in which ϕ is satisfied. A formula is satisfied by a model M, if it is satisfied by one of the states of M. A formula is valid in a model M if it is satisfied by all states of M; and a formula is valid if it is valid in all models.

As mentioned above, we consider programs that terminate abruptly to be non-terminating. Thus, for example, $\langle \texttt{throw x;} \rangle\phi$ is unsatisfiable for all ϕ. Nevertheless, it is possible to express and (if true) prove the fact that a program *p* terminates abruptly. For example, the formula

$$\texttt{e} \doteq \texttt{null} \;\rightarrow\; \langle \texttt{try\{}p\texttt{\}catch(Exception e)\{\}} \rangle (\neg\, (\texttt{e} \doteq \texttt{null}))$$

is true in a state s if and only if the program *p*, when started in s, terminates abruptly by throwing an exception (as otherwise no object is bound to `e`).

Sequents are notated following the scheme

$$\phi_1, \ldots, \psi_m \;\vdash\; \psi_1, \ldots, \psi_n \ ,$$

which has the same semantics as the formula

$$(\forall x_1) \cdots (\forall x_k)((\phi_1 \wedge \ldots \wedge \psi_m) \rightarrow (\psi_1 \vee \ldots \vee \psi_n)) \ ,$$

where $x_1, \ldots, x_k$ are the free variables of the sequent.

4 Sequent Calculus Rules for Handling Abrupt Termination

4.1 Notation

The rules of our calculus operate on the first *active* command $\boldsymbol{p}$ of a program $\pi \boldsymbol{p}\, \omega$. The non-active prefix π consists of an arbitrary sequence of opening braces "`{`", labels, beginnings "`try{`" of `try-catch-finally` blocks, and beginnings "`call(...){`" of method invocation blocks. The prefix is needed to keep track of the blocks that the (first) active command is part of, such that the abruptly terminating statements `throw`, `return`, `break`, and `continue` can be handled appropriately.[2] The postfix ω denotes the "rest" of the program, i.e., everything except the non-active prefix and the part of the program the rule operates on. For example, if a rule is applied to the following JAVA block operating on its first active command `i=0;`, then the non-active prefix π and the "rest" ω are the marked parts of the block:

$$\underbrace{\texttt{l:\{try\{}}_{\pi}\ \texttt{i=0;}\ \underbrace{\texttt{j=0; \}finally\{ k=0; \}\}}}_{\omega}$$

4.2 Loop Rules

Due to space restrictions, we present only one specific rule for `while` loops to demonstrate the properties of loop rules. `for` and `do-while` loops are handled analogously.

The following rule "unwinds" `while` loops. Its application is the prerequisite for symbolically executing the loop body. These "unwind" rules allow to handle `while` loops if used together with induction schemata for the primitive and the user defined types (see the example in Section 5).

$$\frac{\Gamma \vdash (\langle \pi\ \ \texttt{if(}c\texttt{)}\, l'\texttt{:\{}l''\texttt{:\{}p'\texttt{\}}\ \ l_1\texttt{:}\cdots l_n\texttt{:while(}c\texttt{)\{}p\texttt{\}\}}\ \ \omega\rangle\phi)}{\Gamma \vdash (\langle \pi\ \ l_1\texttt{:}\cdots l_n\texttt{:while(}c\texttt{)\{}p\texttt{\}}\ \ \omega\rangle\phi)} \tag{R1}$$

where

- l' and l'' are new labels,
- p' is the result of (simultaneously) replacing in p
  (a) every `break` l_i (for $1 \le i \le n$) and every `break` (with no label) that has the `while` loop as its target by `break` l', and
  (b) every `continue` l_i (for $1 \le i \le n$) and every `continue` (with no label) that has the `while` loop as its target by `break` l''.[3]

The list l_1:, ... , l_n: usually has only one element or is empty, but in general a loop can have more than one label.

In the "unwound" instance p' of the loop body p, the label l' is the new target for `break` statements and l'' is the new target for `continue` statements, which both had

[2] In DL versions for simple artificial programming languages, where no prefixes are needed, any formula of the form $\langle p\ q\rangle\phi$ can be replaced by $\langle p\rangle\langle q\rangle\phi$. In our calculus, splitting of $\langle \pi p q \omega\rangle\phi$ into $\langle \pi p\rangle\langle q\omega\rangle\phi$ is not possible (unless the prefix π is empty) because πp is not a valid program; and the formula $\langle \pi p\, \omega\rangle\langle \pi\, q\, \omega\rangle\phi$ cannot be used either because its semantics is in general different from that of $\langle \pi p q \omega\rangle\phi$.

[3] The target of a `break` or `continue` statement with no label is the loop that immediately encloses it.

the `while` loop as target before. This results in the desired behaviour: `break` abruptly terminates the whole loop, while `continue` abruptly terminates the current instance of the loop body.

A `continue` with or without label is never handled by a rule directly, because it can only occur in loops, where it is always transformed into a break by the loop rules.

4.3 Rules for the Abruptly Terminating Statements

Possible Combinations of Prefix and Abruptly Terminating Statement. In the following, we present rules for combinations of prefix type (beginning of a block, method invocation or `try`) and abruptly terminating statement (`break`, `return` or `throw`). Due to restrictions of the language specification, the combination method invocation/`break` does not occur. Also, `switch` statements, which may contain a `break`, are not considered here; they are transformed into a sequence of `if` statements.

Evaluation of Arguments. The arguments exc and val of statements `throw` exc resp. `return` val must already be evaluated (they must be logical terms) before the appropriate rule for redirecting the control flow can be applied to the abruptly terminating statement. Otherwise, a rule such as the following (rule (R2)) has to be used first, which then allows the application of other rules that evaluate the expression exc.

$$\frac{\Gamma \vdash \langle \pi \ \texttt{\{}x\texttt{=}\mathit{exc}\texttt{; throw } x\texttt{;\}} \ \omega\rangle\phi}{\Gamma \vdash \langle \pi \ \texttt{throw } \mathit{exc}\texttt{;} \ \omega\rangle\phi} \tag{R2}$$

where x is a new variable of the same type as the expression exc. Since, in this paper we focus on the handling of abrupt termination here and not on the evaluation of expressions, we assume in the following that this has already been done.

We also do not consider the problem of undefined expressions in this paper, whose evaluation results in an exception being thrown (e.g., the expression `o.a` if the value of `o` is `null`). If an expression e occurs that may be undefined, the rules have a further premiss $\Gamma \vdash \mathit{isdef}(e)$ in the full version of the calculus.

Rule for Method Call/`return`. The rule for this combination symbolically executes every step the virtual machine does when a method invocation is terminated: The return value is assigned to the location recorded in the method call prefix and `this` is restored to the value it had before method invocation.

$$\frac{\Gamma \vdash \langle \pi \ x\texttt{=}y\texttt{; this=}\mathit{old}\texttt{;} \ \omega\rangle\phi}{\Gamma \vdash \langle \pi \ \texttt{call(}\mathit{old}\texttt{, }x\texttt{):\{return } y\texttt{; } \mathit{pgm}\texttt{\}}\omega\rangle\phi} \tag{R3}$$

In pure Java it is not possible to explicitly assign a value to `this`. Our assignment rule, however, can handle such a statement and produces the desired effect. The "rest" program pgm of the method body, which is not executed, may be empty.

Rule for Method Call/`throw`. In this case, the method is terminated and `this` is restored to its old value, but no return value is assigned. The `throw` statement remains unchanged (i.e., the exception is handed up to the invoking program).

$$\frac{\Gamma \vdash \langle \pi \ \texttt{this=}old\texttt{;}\ \texttt{throw}\ exc\,\texttt{;}\ \omega\rangle\phi}{\Gamma \vdash \langle \pi \ \texttt{method call(}old\texttt{,}\ x\texttt{):\{throw}\ exc\,\texttt{;}\ pgm\texttt{\}}\omega\rangle\phi} \tag{R4}$$

Again, the "rest" pgm of the method body, which is not executed, may be empty.

Rules for `try`/`throw`. The following rules allow to handle `try-catch-finally` blocks and the `throw` statement. These are simplified versions of the actual rules that apply to the case where there is exactly one `catch` clause and one `finally` clause.

$$\frac{\Gamma \vdash \ instanceof(exc, T) \qquad \Gamma \vdash (\langle \pi \ \texttt{try\{}e\texttt{=}exc\texttt{;}q\texttt{\}finally\{}r\texttt{\}}\ \omega\rangle\phi}{\Gamma \vdash (\langle \pi \ \texttt{try\{throw}\ exc\texttt{;}\ p\texttt{\}catch(}T\ e\texttt{)\{}q\texttt{\}finally\{}r\texttt{\}}\ \omega\rangle\phi)} \tag{R5}$$

$$\frac{\Gamma \vdash \neg instanceof(exc, T) \qquad \Gamma \vdash (\langle \pi \ r\texttt{;}\ \texttt{throw}\ exc\texttt{;}\ \omega\rangle\phi)}{\Gamma \vdash (\langle \pi \ \texttt{try\{throw}\ exc\texttt{;}\ p\texttt{\}catch(}T\ e\texttt{)\{}q\texttt{\}finally\{}r\texttt{\}}\ \omega\rangle\phi)} \tag{R6}$$

Rule (R5) applies if an exception exc is thrown that is an instance of exception class T, i.e., the exception is caught; otherwise, if the exception is not caught, rule (R6) applies.

Rules for `try`/`break` and `try`/`return`. A `return` or a `break` statement within a `try-catch-finally` statement causes the immediate execution of the `finally` block. Afterwards the `try` statement terminates abnormally with the `break` resp. the `return` statement (a different abruptly terminating statement in the `finally` block takes precedence). This behaviour is simulated by the following two rules:

$$\frac{\Gamma \vdash \langle \pi \ r\ \texttt{break}\ l\texttt{;}\ \omega\rangle\phi}{\Gamma \vdash \langle \pi \ \texttt{try\{break}\ l\texttt{;}\ p\texttt{\}catch(}T\ exc\texttt{)\{}q\texttt{\}finally\{}r\texttt{\}}\ \omega\rangle\phi} \tag{R7}$$

$$\frac{\Gamma \vdash \langle \pi \ r\ \texttt{return}\ v\texttt{;}\ \omega\rangle\phi}{\Gamma \vdash \langle \pi \ \texttt{try\{return}\ v\texttt{;}\ p\texttt{\}catch(}T\ exc\texttt{)\{}q\texttt{\}finally\{}r\texttt{\}}\ \omega\rangle\phi} \tag{R8}$$

Rules for block/`break`, block/`return`, and block/`throw`. Rules (R9) and (R10) apply to blocks which are terminated by a `break` statement without label resp. with a label l matching one of the labels $l_1, \ldots, l_k$ of the block ($k \geq 0$).

$$\frac{\Gamma \vdash \langle \pi \ \omega\rangle\phi}{\Gamma \vdash \langle \pi \ l_1\texttt{:}\cdots l_k\texttt{:\{break;}\ pgm\texttt{\}}\ \omega\rangle\phi} \tag{R9}$$

$$\frac{\Gamma \vdash \langle \pi \ \omega\rangle\phi}{\Gamma \vdash \langle \pi \ l_1\texttt{:}\cdots l_k\texttt{:\{break}\ l\texttt{;}\ pgm\texttt{\}}\ \omega\rangle\phi} \qquad \text{where } l \in \{l_1, \ldots, l_k\} \tag{R10}$$

The following rules handle labelled and unlabelled blocks that are abruptly terminated by a `break` statement with a label l not matching any of the labels of the block (Rule (R11)), or by a `return` or `throw` statement (Rules (R12) resp. (R13)).

$$\frac{\Gamma \vdash \langle \pi \ \texttt{break} \ l\texttt{;} \ \omega \rangle \phi}{\Gamma \vdash \langle \pi \ l_1\texttt{:} \cdots l_k\texttt{:\{break} \ l\texttt{;} \ pgm\texttt{\}} \ \omega \rangle \phi} \quad \text{where } l \notin \{l_1, \ldots, l_k\} \tag{R11}$$

$$\frac{\Gamma \vdash \langle \pi \ \texttt{return} \ v\texttt{;} \ \omega \rangle \phi}{\Gamma \vdash \langle \pi \ l_1\texttt{:} \cdots l_k\texttt{:\{return} \ v\texttt{;} \ pgm\texttt{\}} \ \omega \rangle \phi} \tag{R12}$$

$$\frac{\Gamma \vdash \langle \pi \ \texttt{throw} \ e\texttt{;} \ \omega \rangle \phi}{\Gamma \vdash \langle \pi \ l_1\texttt{:} \cdots l_k\texttt{:\{throw} \ e\texttt{;} \ pgm\texttt{\}} \ \omega \rangle \phi} \tag{R13}$$

In all the rules above, the program pgm (that is not executed) may be empty.

Rules for Empty Blocks. Rule (R14) applies to empty `try` blocks, which terminate normally. There are similar rules for empty blocks and empty method invocations.

$$\frac{\Gamma \vdash (\langle \pi \ r \ \omega \rangle \phi)}{\Gamma \vdash (\langle \pi \ \texttt{try\{\}catch(}T \ e\texttt{)\{}q\texttt{\}finally\{}r\texttt{\}} \ \omega \rangle \phi)} \tag{R14}$$

5 Example

As an example, we use the calculus presented in the previous section to verify that, if the program

```
while (true) {
  if (i==10) break;
  i++;
}
```

is started in a state in which the value of the variable `i` is between 0 and 10, then it terminates normally in a state in which the value of `i` is 10.[4] That is, we prove that the sequence

$$0 \leq \texttt{i} \land \texttt{i} \leq 10 \ \vdash \ \langle p_{\text{while}} \rangle \texttt{i} \doteq 10 \tag{1}$$

is valid, where p_{while} is an abbreviation for the above while loop. Instead of proving (1) directly, we first use induction to derive the sequence

$$\vdash \ (\forall n)((n \leq 10 \land \texttt{i} \doteq 10 - n) \to \langle p_{\text{while}} \rangle \texttt{i} \doteq 10) \tag{2}$$

as a lemma. It basically expresses the same as (1), the difference is that its form allows a proof by induction on n. The introduction of this lemma is the only step in the proof where an intuition for what the Java Card program p_{while} actually does is needed and where a verification tool may require user interaction.

[4] This example program was presented in [3].

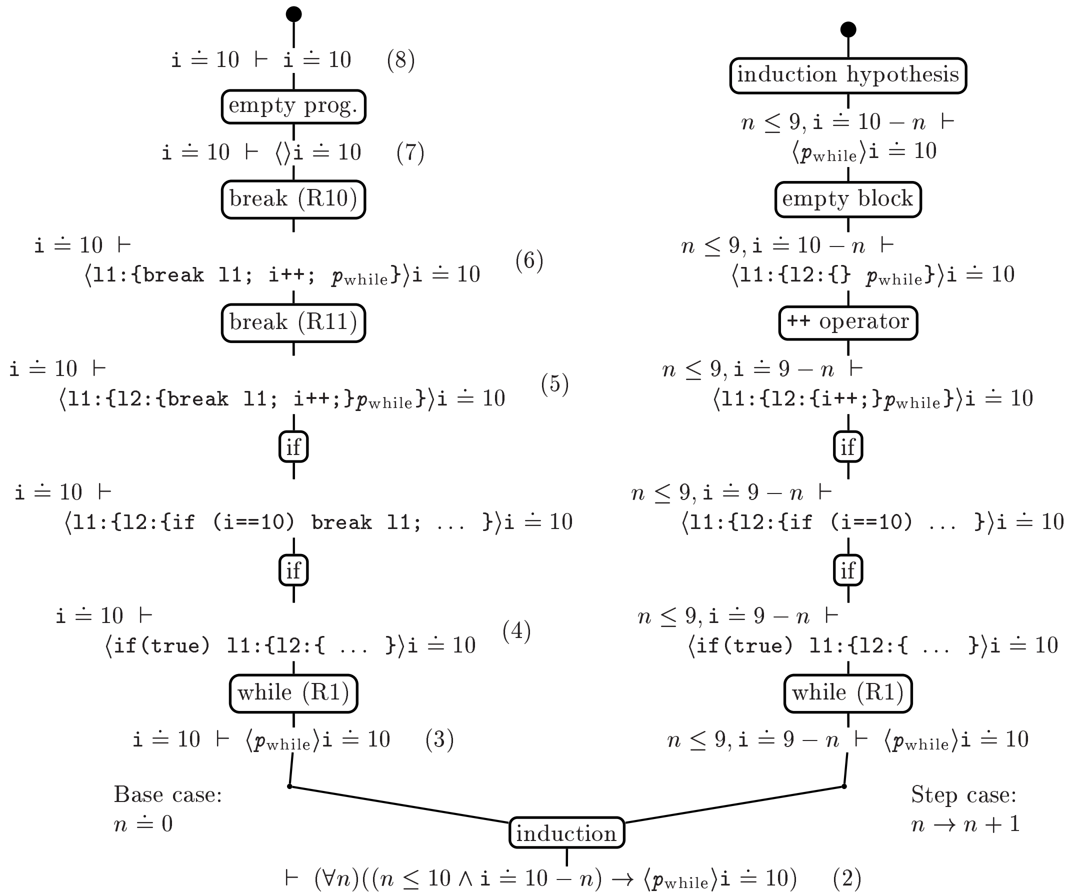

Fig. 1. Structure of the proof for sequent (1).

The derivation of (2) is shown schematically in Figure 1. In the following, we describe the base case $n = 0$ of the induction in detail. The step case is similar (the main difference is that it closes with an application of the induction hypothesis while the base case closes with an axiomatic sequent).

The first sequent which appears in the base case after applying the induction rule and some simplifications is

$$\mathtt{i} \doteq 10 \vdash \langle \mathtt{while\ (true)\ \{if\ (i==10)\ break;\ i++;\}} \rangle \mathtt{i} \doteq 10 \qquad (3)$$

An application of the rule for `while` loops (R1) results in the new proof obligation

$$\begin{array}{l} \mathtt{i} \doteq 10 \vdash \\ \quad \langle \mathtt{if\ (true)\ l1:\{l2:\{if\ (i==10)\ break\ l1;\ i++;\}}\ p_{\mathrm{while}}\mathtt{\}} \rangle \mathtt{i} \doteq 10 \end{array} \qquad (4)$$

Here, two new labels are introduced: `l1` is the target for `break` statements in the loop body and `l2` is the target for `continue` statements (the latter does not occur in this example).

The next step is to use the rule for `if` statements twice. After the second application, we get the sequent

$$\mathtt{i} \doteq 10 \vdash \langle \mathtt{l1:\{l2:\{break\ l1;\ i++\}}\ p_{\mathrm{while}}\mathtt{\}} \rangle \mathtt{i} \doteq 10 \qquad (5)$$

in which the next executable statement is `break l1`. Now, the rule for labelled `break` statements in a block with a non-matching label (R11) has to be applied, which eliminates the block labelled with `l2`:

$$\mathtt{i} \doteq 10 \;\vdash\; \langle \mathtt{l1\!:\!\{break\ l1;}\ p_{\text{while}}\mathtt{\}}\rangle \mathtt{i} \doteq 10 \tag{6}$$

Then, the rule for labelled `break` statements in a block with a *matching* label (R10) is used. The result is

$$\mathtt{i} \doteq 10 \;\vdash\; \langle\rangle(\mathtt{i} \doteq 10) \tag{7}$$

This simplifies with the rule for the empty program to

$$\mathtt{i} \doteq 10 \;\vdash\; \mathtt{i} \doteq 10 \tag{8}$$

and can thus be shown to be valid.

After the lemma (2) has been proved by induction, it can be used to prove the original proof obligation (1). First, we use a quantifier rule to instantiate n with $10 - \mathtt{i}$. The result is

$$0 \leq \mathtt{i} \wedge \mathtt{i} \leq 10 \;\vdash\; (10 - \mathtt{i} \leq 10 \wedge \mathtt{i} \doteq 10 - (\mathtt{i} - 10)) \rightarrow (\langle p_{\text{while}}\rangle \mathtt{i} \doteq 10)$$

which can be simplified to

$$0 \leq \mathtt{i} \wedge \mathtt{i} \leq 10 \wedge \mathtt{i} \doteq \mathtt{i} \;\vdash\; (\langle p_{\text{while}}\rangle \mathtt{i} \doteq 10) \tag{9}$$

And, since (9) is derivable, the original proof obligation (1) is derivable as well, because the trivial equality $\mathtt{i} \doteq \mathtt{i}$ can be omitted.

References

1. Wolfgang Ahrendt, Thomas Baar, Bernhard Beckert, Martin Giese, Elmar Habermalz, Reiner Hähnle, Wolfram Menzel, and Peter H. Schmitt. The KeY approach: Integrating object oriented design and formal verification. In M. Ojeda-Aciego, I. P. de Guzman, G. Brewka, and L. M. Pereira, editors, *Proceedings, Logics in Artificial Intelligence (JELIA), Malaga, Spain*, LNCS 1919. Springer, 2000.
2. Bernhard Beckert. A Dynamic Logic for the formal verification of Java Card programs. In *Proceedings, Java Card Workshop (JCW), Cannes, France*, LNCS 2014. Springer, 2001. To appear. Available at i12www.ira.uka.de/~key.
3. Marieke Huisman and Bart Jacobs. Java program verification via a Hoare logic with abrupt termination. In *Proceedings, Fundamental Approaches to Software Engineering (FASE), Berlin, Germany*, LNCS 1783. Springer, 2000.
4. Dieter Hutter, Bruno Langenstein, Claus Sengler, Jörg H. Siekmann, and Werner Stephan. Deduction in the Verification Support Environment (VSE). In M.-C. Gaudel and J. Woodcock, editors, *Proceedings, International Symposium of Formal Methods Europe (FME), Oxford, UK*, LNCS 1051. Springer, 1996.
5. Dexter Kozen and Jerzy Tiuryn. Logic of programs. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics, chapter 14, pages 789–840. Elsevier, Amsterdam, 1990.
6. Arnd Poetzsch-Heffter and Peter Müller. A programming logic for sequential Java. In S. D. Swierstra, editor, *Proceedings, Programming Languages and Systems (ESOP), Amsterdam, The Netherlands*, LNCS 1576, pages 162–176. Springer, 1999.
7. Wolfgang Reif. The KIV-approach to software verification. In M. Broy and S. Jähnichen, editors, *KORSO: Methods, Languages, and Tools for the Construction of Correct Software – Final Report*, LNCS 1009. Springer, 1995.
8. Kurt Stenzel. Verification of Java Card programs. Technical Report 2001-5, Institut für Informatik, Universität Augsburg, 2001.

Reasoning on UML Class Diagrams in Description Logics

Andrea Calì, Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini

Dipartimento di Informatica e Sistemistica
Università di Roma "La Sapienza"
Via Salaria 113, I-00198 Roma, Italy
lastname@dis.uniroma1.it

Abstract. In this paper[1] we formalize UML class diagrams in terms of a logic belonging to Description Logics, which are subsets of First-Order Logic that have been thoroughly investigated in Knowledge Representation. The logic we have devised is specifically tailored towards the high expressiveness of UML information structuring mechanisms, and allows one to formally model important properties which typically can only be specified by means of qualifiers. The logic is equipped with decidable reasoning procedures which can be profitably exploited in reasoning on UML class diagrams. This makes it possible to provide computer aided support during the application design phase in order to automatically detect relevant properties, such as inconsistencies and redundancies.

1 Introduction

The *Unified Modeling Language* (UML) is the de facto standard formalism for object-oriented modeling [2, 14]. There is a vast consensus on the need for a precise semantics for UML [12, 17], in particular for UML class diagrams. Indeed, several types of formalization of UML class diagrams have been proposed in the literature [11–13, 9]. Many of them have been proved very useful with respect to the task of establishing a common understanding of the formal meaning of UML constructs. However, to the best of our knowledge, none of them has the explicit goal of building a solid basis for allowing automated reasoning techniques, based on algorithms that are sound and complete wrt the semantics, to be applicable to UML class diagrams.

In this paper, we propose a new formalization of UML class diagrams in terms of a particular formal logic of the family of Description Logics (DLs). DLs[2] have been proposed as successors of semantic network systems like KL-ONE, with an explicit model-theoretic semantics. The research on these logics has resulted in a number of automated reasoning systems [18, 19, 15, 16], that have been successfully tested in various application domains (see e.g., [21, 22, 20]). Our goal is to exploit the deductive capabilities of DL systems, and show that effective reasoning can be carried out on UML class diagrams, so as to provide support during the specification phase of software development.

In DLs, the domain of interest is modeled by means of *concepts* and *relations*, which denote classes of objects and relation between objects, respectively. Generally speaking, a DL is formed by three basic components:

[1] A full version of this paper can be found in [3].
[2] See http://dl.kr.org for the home page of Description Logics.

– A *description language*, which specifies how to construct complex concept and relationship expressions (also called simply concepts and relationships), by starting from a set of atomic symbols and by applying suitable constructors,
– a *knowledge specification mechanism*, which specifies how to construct a DL knowledge base, in which properties of concepts and relationships are asserted, and
– a set of *automatic reasoning procedures*, which are sound, complete and terminating.

The set of allowed constructors characterizes the expressive power of the description language. Various languages have been considered by the DL community, and numerous papers investigate the relationship between expressive power and computational complexity of reasoning (see [10] for a survey).

Several works point out that DLs can be profitably used to provide both formal semantics and reasoning support to formalisms in areas such as Natural Language, Configuration Management, Database Management, Software Engineering. For example, [7, 8] illustrates the use of DLs for database modeling. However, DLs have not been applied to the Unified Modeling Language (UML) (with the exception of [5]). In this work we concentrate on UML class diagrams for the conceptual perspective. Hence, we do not deal with those features that are relevant for the implementation perspective, such as public, protected, and private qualifiers for methods and attributes. For such UML class diagrams we present a formalization of UML in terms of DLs. In particular, we show how to capture the constructs of UML class diagrams by using a Description Logic that is equipped with n-ary relations. The DL we have adopt is specifically tailored towards the high expressiveness of UML information structuring mechanisms, and allows one to formally model important additional properties, such has disjointness of classes, or partitions of classes into subclasses, that are typically specified by means of constraints in UML class diagrams. In spite of the expressiveness required, the logic proposed admits decidable reasoning procedures. Overall, the formalization in DLs of UML class diagrams provides us with a rigorous logical framework for representing and automatically reasoning on UML class specifications. Such a formalization can be considered as the basic steps towards developing intelligent tools that provide computer aided reasoning support during the application design phase, in order to automatically detect relevant properties, such as inconsistencies and redundancies.

The paper is organized as follows: in Section 2 we give an overview of the Description Logic we use, called $\mathcal{DLR}$. In Sections 3, 4, 5 and 6, we illustrate the formalization of UML class diagrams in terms of $\mathcal{DLR}$, focusing on classes, associations, generalization, and constraints, respectively. In Section 7 we discuss the use of the reasoning procedures associated to $\mathcal{DLR}$ in order to support the specification of UML class diagrams. Section 8 concludes the paper.

2 The Description Logic $\mathcal{DLR}$

In this paper we adopt a DL, here called $\mathcal{DLR}$, presented in [6], which is a variant of logic originally introduced in [4]. The basic elements of $\mathcal{DLR}$ are *concepts* (unary relations), and *n-ary relations*. We assume to deal with a finite set of atomic relations and atomic concepts, denoted by P and A, respectively. Arbitrary relations (of given arity between 2 and n_{max}), denoted by R, and arbitrary concepts, denoted by C, are built according to the following syntax:

$$R ::= \top_n \mid P \mid (i/n : C) \mid \neg R \mid R_1 \sqcap R_2$$
$$C ::= \top_1 \mid A \mid \neg C \mid C_1 \sqcap C_2 \mid (\leq k\,[i]R)$$

where i denotes a component of a relation, i.e., an integer between 1 and n_{max}, n denotes the *arity* of a relation, i.e., an integer between 2 and n_{max}, and k denotes a non-negative integer. We consider only concepts and relations that are *well-typed*, which means that (i) only relations of the same arity n are combined to form expressions of type $R_1 \sqcap R_2$ (which inherit the arity n), and (ii) $i \leq n$ whenever i denotes a component of a relation of arity n.

We also make use of the following abbreviations:

$$\begin{array}{rcl} C_1 \sqcup C_2 & \text{for} & \neg(\neg C_1 \sqcap \neg C_2) \\ C_1 \Rightarrow C_2 & \text{for} & \neg C_1 \sqcup C_2 \\ (\geq k\,[i]R) & \text{for} & \neg(\leq k{-}1\,[i]R) \\ \exists[i]R & \text{for} & (\geq 1\,[i]R) \\ \forall[i]R & \text{for} & \neg\exists[i]\neg R \end{array}$$

Moreover, we abbreviate $(i/n : C)$ with $(i : C)$, when n is clear from the context.

A $\mathcal{DLR}$ *knowledge base* (KB) is constituted by a finite set of *inclusion assertions*, where each assertion has one of the forms:

$$R_1 \sqsubseteq R_2 \qquad\qquad C_1 \sqsubseteq C_2$$

with R_1 and R_2 of the same arity.

Besides inclusion assertions, $\mathcal{DLR}$ KBs allow for assertions expressing identification constraints and functional dependencies.

An *identification assertion* on a concept has the form:

$$(\textbf{id}\ C\ [i_1]R_1, \ldots, [i_h]R_h)$$

where C is a concept, each R_j is a relation, and each i_j denotes one component of R_j. Intuitively, such an assertion states that no two different instances of C agree on the participation to $R_1, \ldots, R_h$. In other words, if a is an instance of C that is the i_j-th component of a tuple t_j of R_j, for $j \in \{1, \ldots, h\}$, and b is an instance of C that is the i_j-th component of a tuple s_j of R_j, for $j \in \{1, \ldots, h\}$, and for each j, t_j agrees with s_j in all components different from i_j, then a and b coincide.

A *functional dependency assertion* on a relation has the form:

$$(\textbf{fd}\ R\ i_1, \ldots, i_h \rightarrow j)$$

where R is a relation, $h \geq 2$, and $i_1, \ldots, i_h, j$ denote components of R. The assertion imposes that two tuples of R that agree on the components $i_1, \ldots, i_h$, agree also on the component j.

Note that unary functional dependencies (i.e., functional dependencies with $h = 1$) are ruled out in $\mathcal{DLR}$, since these lead to undecidability of reasoning [6]. Note also that the right hand side of a functional dependency contains a single element. However, this

$$
\begin{array}{rlrl}
\top_n^{\mathcal{I}} & \subseteq (\Delta^{\mathcal{I}})^n & \top_1^{\mathcal{I}} & = \Delta^{\mathcal{I}} \\
P^{\mathcal{I}} & \subseteq \top_n^{\mathcal{I}} & A^{\mathcal{I}} & \subseteq \Delta^{\mathcal{I}} \\
(i/n : C)^{\mathcal{I}} & = \{t \in \top_n^{\mathcal{I}} \mid t[i] \in C^{\mathcal{I}}\} & (\neg C)^{\mathcal{I}} & = \Delta^{\mathcal{I}} \setminus C^{\mathcal{I}} \\
(\neg R)^{\mathcal{I}} & = \top_n^{\mathcal{I}} \setminus R^{\mathcal{I}} & (C_1 \sqcap C_2)^{\mathcal{I}} & = C_1^{\mathcal{I}} \cap C_2^{\mathcal{I}} \\
(R_1 \sqcap R_2)^{\mathcal{I}} & = R_1^{\mathcal{I}} \cap R_2^{\mathcal{I}} & (\leq k\,[i]R)^{\mathcal{I}} & = \{a \in \Delta^{\mathcal{I}} \mid \sharp\{t \in R_1^{\mathcal{I}} \mid t[i] = a\} \leq k\}
\end{array}
$$

Fig. 1. Semantic rules for $\mathcal{DLR}$ (P, R, R_1, and R_2 have arity n)

is not a limitation, because any functional dependency with more than one element in the right hand side can always be split into several dependencies of the above form.

The semantics of $\mathcal{DLR}$ is specified through the notion of interpretation. An *interpretation* $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ of a $\mathcal{DLR}$ KB $\mathcal{K}$ is constituted by an *interpretation domain* $\Delta^{\mathcal{I}}$ and an *interpretation function* $\cdot^{\mathcal{I}}$ that assigns to each concept C a subset $C^{\mathcal{I}}$ of $\Delta^{\mathcal{I}}$ and to each relation R of arity n a subset $R^{\mathcal{I}}$ of $(\Delta^{\mathcal{I}})^n$, such that the conditions in Figure 1 are satisfied. (In the figure, $t[i]$ denotes the i-th component of tuple t.) We observe that $\top_1$ denotes the interpretation domain, while $\top_n$, for $n > 1$, does *not* denote the n-Cartesian product of the domain, but only a subset of it, that covers all relations of arity n. It follows, from this property, that the "$\neg$" constructor on relations is used to express difference of relations, rather than complement.

To specify the semantics of a KB we first define when an interpretation satisfies an assertion as follows:

– An interpretation $\mathcal{I}$ *satisfies* an inclusion assertion $R_1 \sqsubseteq R_2$ (resp. $C_1 \sqsubseteq C_2$) if $R_1^{\mathcal{I}} \subseteq R_2^{\mathcal{I}}$ (resp. $C_1^{\mathcal{I}} \subseteq C_2^{\mathcal{I}}$).
– An interpretation $\mathcal{I}$ *satisfies* the assertion (**id** C $[i_1]R_1, \ldots, [i_h]R_h$) if for all $a, b \in C^{\mathcal{I}}$ and for all $t_1, s_1 \in R_1^{\mathcal{I}}, \ldots, t_h, s_h \in R_h^{\mathcal{I}}$ we have that:

$$
\left.\begin{array}{l}
a = t_1[i_1] = \cdots = t_h[i_h], \\
b = s_1[i_1] = \cdots = s_h[i_h], \\
t_j[i] = s_j[i], \text{ for } j \in \{1, \ldots, h\}, \text{ and for } i \neq i_j
\end{array}\right\} \text{ implies } a = b
$$

– An interpretation $\mathcal{I}$ *satisfies* the assertion (**fd** R $i_1, \ldots, i_h \to j$) if for all $t, s \in R^{\mathcal{I}}$, we have that:

$$
t[i_1] = s[i_1], \ \ldots, \ t[i_h] = s[i_h] \quad \text{implies} \quad t[j] = s[j]
$$

An interpretation that satisfies all assertions in a KB $\mathcal{K}$ is called a *model* of $\mathcal{K}$.

Several reasoning services are applicable to $\mathcal{DLR}$ KBs. The most important ones are KB satisfiability and logical implication. A KB $\mathcal{K}$ is *satisfiable* if there exists a model of $\mathcal{K}$. A concept C is *satisfiable* in a KB $\mathcal{K}$ if there is a model $\mathcal{I}$ of $\mathcal{K}$ such that $C^{\mathcal{I}}$ is nonempty. A concept C_1 is *subsumed by* a concept C_2 in a KB $\mathcal{K}$ if $C_1^{\mathcal{I}} \subseteq C_2^{\mathcal{I}}$ for every model $\mathcal{I}$ of $\mathcal{K}$. An assertion α is *logically implied* by $\mathcal{K}$ if all models of $\mathcal{K}$ satisfy α. One can easily verify that logical implication and KB unsatisfiability are mutually reducible.

One of the distinguishing features of $\mathcal{DLR}$ is that it is equipped with reasoning algorithms that are sound and complete wrt to the semantics. Such algorithms allow one to decide all the above reasoning tasks in deterministic exponential time [6]. Indeed, the proposed algorithms are computationally optimal, since reasoning in $\mathcal{DLR}$ is EXPTIME-complete [4].

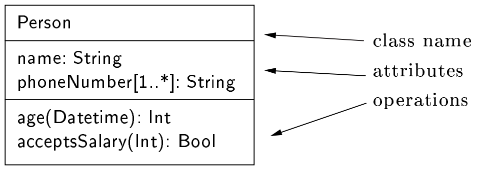

Fig. 2. Representation of a class in UML

3 Classes

A *class* in an UML class diagram denotes a *sets of objects* with common features. A class is graphically rendered as a rectangle divided into three parts, as shown for example in Figure 2. The first part contains the *name* of the class, which has to be unique in the whole diagram. The second part contains the *attributes* of the class, each denoted by a name (possibly followed by the *multiplicity*, between square brackets) and with an associated *class*, which indicates the domain of the attribute values. For example, the attribute phoneNumber[1..*]: String means that each instance of the class has at least one phone number, and possibly more, and that each phone numbers is an instance of String. If not otherwise specified, attributes are *single-valued*. The third part contains the *operations* of the class, i.e., the operations associated to the objects of the class. An operation definition has the form:

$$\textit{operation-name}(\textit{parameter-list}) \colon (\textit{return-list})$$

Observe that an operation may return a *tuple* of objects as result.

An UML class is represented by a $\mathcal{DLR}$ concept. This follows naturally from the fact that both UML classes and $\mathcal{DLR}$ concepts denote *sets of objects*.

An UML *attribute* a of type C' for a class C associates to each instance of C, zero, one, or more instances of a class C'. An optional *multiplicity* $[i..j]$ for a specifies that a associates to each instance of C, at least i and most j instances of C'. When the multiplicity is missing, [1..1] is assumed, i.e., the attribute is *mandatory* and *single-valued*.

To formalize attributes we have to think of an attribute a of type C' for a class C as a binary relation between instances of C and instances of C'. We capture such a binary relation by means of a binary relation a of $\mathcal{DLR}$. To specify the type of the attribute we use the assertion:

$$C \ \sqsubseteq \ \forall[1](a \Rightarrow (2 : C'))$$

Such an assertion specifies precisely that, for each instance c of the concept C, all objects related to c by a, are instances of C'. Note that an attribute name is not necessarily unique in the whole schema, and hence two different classes could have the same attribute, possibly of different types. This situation is correctly captured by the formalization in $\mathcal{DLR}$.

To specify the multiplicity $[i..j]$ associated to the attribute we add the assertion:

$$C \ \sqsubseteq (\geq i\,[1]a) \sqcap (\leq j\,[1]a)$$

Such an assertion specifies that each instance of C participates at least i times and at most j times to relation a via component 1. If $i = 0$, i.e., the attribute is *optional*, we omit the first conjunct, and if $j = *$ we omit the second one.

An operation of a class is a function from the objects of the class to which the operation is associated, and possibly additional parameters, to tuples of objects. In class diagrams, the code associated to the operation is not considered and typically, what is represented is only the signature of the operation.

In $\mathcal{DLR}$, we model operations by means of $\mathcal{DLR}$ relations. Let

$$f(P_1, \ldots, P_m) : (R_1, \ldots, R_n)$$

be an operation of a class C that has m parameters belonging to the classes $P_1, \ldots, P_m$ respectively and n return values belonging to $R_1, \ldots, R_n$ respectively. We formalize such an operation as a $\mathcal{DLR}$ relation, named $\mathtt{op}_{f(P_1,\ldots,P_m):(R_1,\ldots,R_n)}$, of arity $m+n+1$ among instances of the $\mathcal{DLR}$ concepts $C, P_1, \ldots, P_m, R_1, \ldots, R_n$. On such a relation we enforce the following assertions:

– An assertion imposing the correct types to parameters and return values:

$$\begin{aligned} C \ \sqsubseteq\ & \forall[1](\mathtt{op}_{f(P_1,\ldots,P_m):(R_1,\ldots,R_n)} \Rightarrow \\ & ((2\!:\!P_1) \sqcap \cdots \sqcap (m+1\!:\!P_m) \sqcap (m+2\!:\!R_1) \sqcap \cdots \sqcap (m+n+1\!:\!R_n)) \end{aligned}$$

– Assertions imposing that invoking the operation on a given object with given parameters determines in a unique way each return value (i.e., the relation corresponding to the operation is in fact a function from the invocation object and the parameters to the returned values):

$$\begin{gathered} (\mathbf{fd}\ \mathtt{op}_{f(P_1,\ldots,P_m):(R_1,\ldots,R_n)}\ 1, \ldots, m+1 \rightarrow m+2) \\ \cdots \\ (\mathbf{fd}\ \mathtt{op}_{f(P_1,\ldots,P_m):(R_1,\ldots,R_n)}\ 1, \ldots, m+1 \rightarrow m+n+1) \end{gathered}$$

These functional dependencies are determined only by the number of parameters and the number of result values, and not by the specific class for which the operation is defined, nor by the types of parameters and result values.

The *overloading* of operations does not pose any difficulty in the formalization since an operation is represented in $\mathcal{DLR}$ by a relation having as name the whole signature of the operation, which consists not only the name of the operation but also the parameter and return value types. Observe that the formalization of operations in $\mathcal{DLR}$ correctly allows one to have operations with the same name or even with the same signature in two different classes.

4 Associations and Aggregations

An *association* in UML, graphically rendered as in Figure 3, is a relation between the instances of two or more classes. An association often has a related *association class* that describes properties of the association such as attributes, operations, etc. An *aggregation* in UML, graphically rendered as in Figure 4, is a binary relation between the instances of two classes, denoting a part-whole relationship, i.e., a relationship that specifies that each instance of a class is made up of a set of instances of another class.

Observe that names of associations and names of aggregations (as names of classes) are *unique*. In other words there cannot be two associations/aggregations with the same name.

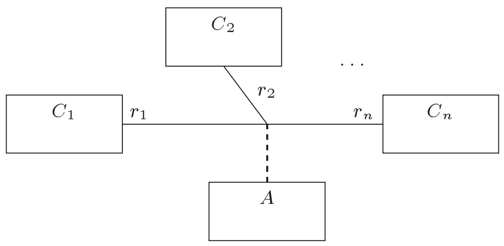

Fig. 3. Association in UML

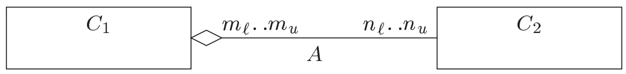

Fig. 4. Aggregation in UML

We first concentrate on the formalization of aggregations, which are simpler to model than general associations. An aggregation A, saying that instances of the class C_1 have components that are instances of the class C_2, is formalized in $\mathcal{DLR}$ by means of a binary relation A together with the following assertion:

$$A \sqsubseteq (1\!:\!C_1) \sqcap (2\!:\!C_2).$$

Note that the distinction between the contained class and the containing class is not lost. Indeed, we simply use the following convention: *the first argument of the relation is the containing class.*

As we have seen for class attributes, the multiplicity of an aggregation can be easily expressed in $\mathcal{DLR}$. For example, the multiplicities shown in Figure 4 are formalized by means of the assertions:

$$\begin{array}{lcl} C_1 & \sqsubseteq & (\geq n_\ell\,[1]A) \sqcap (\leq n_u\,[1]A) \\ C_2 & \sqsubseteq & (\geq m_\ell\,[2]A) \sqcap (\leq m_u\,[2]A) \end{array}$$

We can use a similar assertion for a multiplicity on the participation of instances of C_1 for each given instance of C_2.

Observe that, in the formalization in $\mathcal{DLR}$ of aggregation, role names do not play any role. If we want to keep track of them in the formalization, it suffices to consider them as convenient abbreviations for the components of the $\mathcal{DLR}$ relation modeling the aggregation.

The decision of representing an aggregation by a binary $\mathcal{DLR}_{ifd}$ relation leads some implications; first of all, we note that role names are lost. In our framework role names are replaced by an integer i (whose value can be only 1 or 2), which specifies whether the corresponding argument is the first or the second in the aggregation.

Now, we want to preserve the role names in our framework. Therefore we take advantage of the *set of role names* $\mathcal{N}$ introducing, for each *atomic* relation R of arity k, a *role name function*

$$f_R : \{1, \ldots, k\} \longrightarrow \mathcal{N} \cup \{\varepsilon\}.$$

The function f_R returns, given an integer i between 1 and k (the arity of the relation), the role name associated to the i-th role, if it has one, ε if the role has no name.

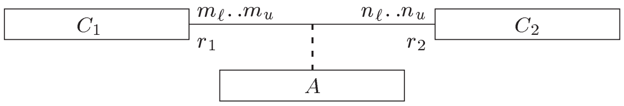

Fig. 5. Binary association in UML

When we compose two relations with the the operator $\sqcap$ (we recall that the relations have got to have the same arity), role name are preserved if both overlapping roles have the same name in $\mathcal{N}$; otherwise, the role names are lost. Formally:

$$f_{R_1 \sqcap R_2}(i) = \begin{cases} f_{R_1}(i), \text{ if } f_{R_1}(i) = f_{R_2}(i) \\ \varepsilon, \qquad \text{otherwise} \end{cases}$$

The negation of a relation R of arity k retains all the role names of the original relation. This choice could seem insensible at a first glance, but it ensures for example that matching role names are preserved when we do the union of relations (like $R_1 \sqcup R_2 = \neg(\neg R_1 \sqcap \neg R_2)$). Formally we have:

$$f_{\neg R}(i) = f_R(i) \text{ for each } i \in \{1, \ldots, k\} \tag{1}$$

We impose f_R to be *injective* for every relation R; instead, we would have the same name for more than one role, within the same relation.

Next we focus on *associations*. Since associations have often a related association class, we formalize associations in $\mathcal{DLR}$ by reifying each association A into a $\mathcal{DLR}$ concept A with suitable properties. We represent an association among n classes $C_1, \ldots, C_n$, as shown in Figure 3, by introducing a concept A and n *binary* relations $r_1, \ldots, r_n$, one for each component of the association A [3]. Each binary relation r_i has C_i as its first component and A as its second component. Then we enforce the following assertion:

$$\begin{array}{rcl} C & \sqsubseteq & \exists[1]r_1 \sqcap (\leq 1\,[1]r_1) \sqcap \forall[1](r_1 \Rightarrow (2:C_1)) \sqcap \\ & & \exists[1]r_2 \sqcap (\leq 1\,[1]r_2) \sqcap \forall[1](r_2 \Rightarrow (2:C_2)) \sqcap \\ & & \quad \vdots \\ & & \exists[1]r_n \sqcap (\leq 1\,[1]r_n) \sqcap \forall[1](r_n \Rightarrow (2:C_n)) \end{array}$$

where $\exists[1]r_i$ (with $i \in \{1, \ldots, n\}$) specifies that the concept A must have all components $r_1, \ldots, r_n$ of the association A, $(\leq 1\,[1]r_i)$ (with $i \in \{1, \ldots, n\}$) specifies that each such component is single-valued, and $\forall[1](r_i \Rightarrow (2:C_i))$ (with $i \in \{1, \ldots, n\}$) specifies the class each component has to belong to. Finally, we use the assertion

$$(\mathbf{id}\ A\ [1]r_1, \ldots, [1]r_n)$$

to specify that each instance of the concept A indeed represents a *distinct* tuple of the corresponding association.

We can easily represent a multiplicity on a binary UML association, by imposing suitable number restrictions on the $\mathcal{DLR}$ relations modeling the components of the

[3] These relations may have the name of the roles of the association if available in the UML diagram, or an arbitrary name if role names are not available. In any case, we preserve the possibility of using the same role name in different associations.

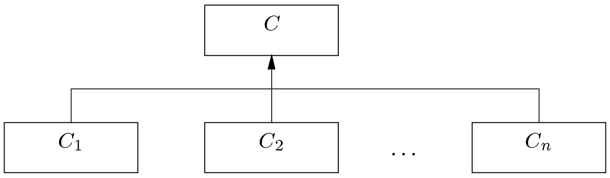

Fig. 6. A class hierarchy in UML

association. Differently from aggregation, however, the names of such relations (which correspond to roles) are unique wrt to the association only, not the entire diagram. Hence we have to state such constraints in $\mathcal{DLR}$ in a slightly different way.

The multiplicities shown in Figure 5 are captured as follows:

$$\begin{array}{rcl} C_1 & \sqsubseteq & (\geq n_\ell\,[1](r_1 \sqcap (2\!:\!A))) \sqcap (\leq n_u\,[1](r_1 \sqcap (2\!:\!A))) \\ C_2 & \sqsubseteq & (\geq m_\ell\,[1](r_2 \sqcap (2\!:\!A))) \sqcap (\leq m_u\,[1](r_2 \sqcap (2\!:\!A))) \end{array}$$

5 Generalization and Inheritance

In UML one can use *generalization* between a parent class and a child class to specify that each instance of the child class is also an instance of the parent class. Hence, the instances of the child class inherit the properties of the parent class, but typically they satisfy additional properties that do not hold for the parent class.

Generalization is naturally supported in $\mathcal{DLR}$. If an UML class C_2 generalizes a class C_1, we can express this by the $\mathcal{DLR}$ assertion:

$$C_1 \sqsubseteq C_2$$

Inheritance between $\mathcal{DLR}$ concepts works exactly as inheritance between UML classes. This is an obvious consequence of the semantics of $\sqsubseteq$ which is based on subsetting. Indeed, in $\mathcal{DLR}$, given an assertion $C_1 \sqsubseteq C_2$, every tuple in a relation having C_2 as i-th argument type may have as i-th component an instance of C_1, which is in fact also an instance of C_2. As a consequence, in the formalization, each attribute or operation of C_2, and each aggregation and association involving C_2 is correctly inherited by C_1. Observe that the formalization in $\mathcal{DLR}$ also captures directly inheritance among association classes, which are treated exactly as all other classes, and multiple inheritance between classes (including association classes).

Moreover in UML, one can group several generalizations into a class hierarchy, as shown in Figure 6. Such a hierarchy is captured in $\mathcal{DLR}$ by a set of inclusion assertions, one between each child class and the parent class:

$$C_i \sqsubseteq C \qquad \text{for each } i \in \{1, \ldots, n\}$$

In UML it is possible to override attributes or operations of a superclass. That is, it is possible to specialize an attribute or an operation for the subclass. From the conceptual point of view such a specialization needs to remain compatible with the original definition of the attribute/operation, i.e., the attribute/operation of the subclass can only be a *restriction* of the corresponding attribute/operation belonging to the superclass. For attributes, this means that one can restrict the type of the attribute to be a subclass of the original type, or restrict the multiplicity wrt to the one specified for the superclass. For operations, while keeping the same signature, one may restrict (by means of constraints) the return types and possibly also the argument types to be subclasses of the original ones[4].

We illustrate by means of an example how one can correctly model such forms of overriding in $\mathcal{DLR}$. Let C be an UML class that has an operation $f(C_1, C_2) : C_3$, and C' be a subclass of C (and hence inherits the operation). In $\mathcal{DLR}$, we model the situation by introducing a concept C and a relation $\mathtt{op}_{f(C_1,C_2):C_3}$ and a concept C' with suitable assertions including $C' \sqsubseteq C$. As a consequence instances of the concept C' inherits the properties that hold for instances of C including the participation in the relation $\mathtt{op}_{f(C_1,C_2):C_3}$. Suppose now that in the UML class diagram C' we override the method $f(C_1, C_2) : C_3$ by requiring that the result value belongs to a subclass C_3' of C_3. We can capture this in $\mathcal{DLR}$ by adding the assertion:

$$C' \ \sqsubseteq \ \forall[1](\mathtt{op}_{f(C_1,C_2):C_3} \Rightarrow (4 : C_3'))$$

6 Constraints

In UML it is possible to add information to a class diagram by using *constraints*. In general, constraints are used to express in an informal way information which cannot be expressed by other constructs of UML class diagrams. We discuss here common types of constraints that occur in UML class diagrams and how they can be taken into account when formalizing class diagrams in $\mathcal{DLR}$.

Generally, in UML class diagrams, unless specified otherwise by a constraint, two classes may have common instances, i.e., they are *not disjoint*. If a constraint imposes the disjointness of two classes, say C and C', this can be formalized in $\mathcal{DLR}$ by means of the assertion

$$C \ \sqsubseteq \ \neg C'$$

Observe that disjointness constraints are often used in class hierarchies. For example, consider a class hierarchy formed by a class C and n subclasses of C, $C_1, \ldots, C_n$. We may want to require that $C_1, \ldots, C_n$ are *mutually disjoint*. In $\mathcal{DLR}$, this can be expressed by the assertions

$$C_i \ \sqsubseteq \ \neg C_j \qquad \text{for each } i, j \in \{1, \ldots, n\} \text{ with } i \neq j$$

Disjointness of classes is just one example of *negative information*. Again, by exploiting the expressive power of $\mathcal{DLR}$, we can express additional forms of negative information, usually not considered in UML, by introducing suitable assertions. For example, we can enforce that no instance of a class C has an attribute a by means of the assertion

$$C \ \sqsubseteq \ \neg\exists[1]a$$

Analogously, one can assert that no instance of a class is involved in a given association or aggregation.

[4] Observe that restricting the argument types corresponds, in the implementation of the operation, to restrict the preconditions for the applicability of the operation.

Turning again the attention to generalization hierarchies, by default, in UML a generalization hierarchy is open, in the sense that there may be instances of the superclass that are not instances of any of the subclasses. This allows for extending the schema more easily, in the sense that the introduction of a new subclass does not change the semantics of the superclass. However, in specific situations, it may happen that in a generalization hierarchy, the superclass C is a covering of the subclasses $C_1, \ldots, C_n$. We can represent such a situation in $\mathcal{DLR}$ by simply including the additional assertion

$$C \sqsubseteq C_1 \sqcup \cdots \sqcup C_n$$

The above assertion models a form of *disjunctive information*: each instance of C is either an instance of C_1, or an instance of C_2, ... or an instance of C_n. Other forms of disjunctive information can be modeled by exploiting the expressive power of $\mathcal{DLR}$. For example, that an attribute a is present only for a specified set $C_1, \ldots, C_n$ of classes can be modeled by suitably using union of classes as follows:

$$\exists[1]a \sqsubseteq C_1 \sqcup \cdots \sqcup C_n$$

Keys are a modeling notion that is very common in databases, and they are used to express that certain attributes uniquely identify the instances of a class. We can exploit the expressive power of $\mathcal{DLR}$ in order to associate keys to classes. If an attribute a is a key for a class C this means that there is no pair of instances of C that have the same value for a. We can capture this in $\mathcal{DLR}$ by means of the assertion $(\mathbf{id}\ C\ [1]a)$. More generally, we are able to specify that a *set* of attributes $\{a_1, \ldots, a_n\}$ is a key for C; in this case we use the assertion: $(\mathbf{id}\ C\ [1]a_1, \ldots, [1]a_n)$

As already seen, constraints that correspond to the specialization of the type of an attribute or its multiplicity can be represented in $\mathcal{DLR}$. Similarly, consider the case of a class C participating in an aggregation A with a class D, and where C and D have subclasses C' and D' respectively, related via an aggregation A'. A *subset constraint* from A' to A can be modeled correctly in $\mathcal{DLR}$ by means of the assertion $A \sqsubseteq A'$, involving the two binary relations A and A' that represent the aggregations.

In general, one can exploit the expressive power of $\mathcal{DLR}$ to formalize several types of constraints that allow one to better represent the application semantics and that are typically not dealt with in a formal way. Observe that this allows one to take such constraints fully into account when reasoning on the class diagram.

7 Reasoning on Class Diagrams

Traditional CASE tools support the designer with a user-friendly graphical environment and provide powerful means to access different kinds of repositories that store information associated to the elements of the developed project. However, no support for higher level activities related to managing the complexity of the design is provided. In particular, the burden of checking relevant properties of class diagrams, such as consistency or redundancy, is left to the responsibility of the designer. Thus, the formalization in $\mathcal{DLR}$ of UML class diagrams, and the fact that properties of inheritance and relevant types of constraints are perfectly captured by the formalization in $\mathcal{DLR}$ and the associated reasoning tasks, provide the ability to reason on class diagrams. This represents a significant improvement and it is a first step towards the development of modeling tools that offer an automated reasoning support to the designer in his modeling activity. By exploiting the $\mathcal{DLR}$ reasoning services various kinds of checks can be performed on the class diagram.

A class diagram is *consistent*, if its classes can be populated without violating any of the constraints in the diagram. Observe that the interaction of various types of constraints may make it very difficult to detect inconsistencies. A *class* is *consistent* if it can be populated without violating any of the constraints in the class diagram. The inconsistency of a class may be due to a design error or due to over-constraining. In any case, the designer can be forced to remove the inconsistency, either by correcting the error, or by relaxing some constraints, or by deleting the class, thus removing redundancy from the schema. By exploiting the formalization in $\mathcal{DLR}$, class consistency can be checked by verifying satisfiability of the corresponding concept in the $\mathcal{DLR}$ KB representing the class diagram. Similarly, consistency of the class diagram corresponds to consistency of the $\mathcal{DLR}$ KB.

Two classes are *equivalent* if they denote the same set of instances whenever the constraints imposed by the class diagram are satisfied. Determining equivalence of two classes allows for their merging, thus reducing the complexity of the schema. A class C_1 is *subsumed by* a class C_2 if, whenever the constraints imposed by the class diagram are satisfied, the extension of C_1 is a subset of the extension of C_2. Such a subsumption allows one to deduce that properties for C_1 hold also for C_2. It is also the basis for a *classification* of all the classes in a diagram. Such a classification, as in any object-oriented approach, can be exploited in several ways within the modeling process [1]. Class equivalence, subsumption, and hence classification, can be checked by verifying equivalence and subsumption in $\mathcal{DLR}$.

A property is a *logical consequence* of a class diagram if it holds whenever all constraints specified in the diagram are satisfied. As an example, consider a class C generalizing classes $C_1, \ldots, C_n$, and assume that a constraint specifies that it is complete. If an attribute a is defined as mandatory for all classes $C_1, \ldots, C_n$, then it follows logically that the same attribute is mandatory also for class C, even if not explicitly present in the schema. Determining logical consequence is useful on the one hand to reduce the complexity of the schema by removing those constraints that logically follow from other ones, and on the other hand it can be used to make properties explicit that are implicit in the schema, thus enhancing its readability. Logical consequence can be captured by logical implication in $\mathcal{DLR}$, and determining logical implication is at the basis of all types of reasoning that a $\mathcal{DLR}$ reasoning system can provide. In particular, observe that all reasoning tasks we have considered above can be rephrased in terms of logical consequence.

8 Conclusions

We have proposed a new formalization of UML class diagrams in terms of a particular formal logic of the family of Description Logics. Notably such a logic has sound, complete and decidable reasoning procedures. These reasoning procedures can be favorably exploited for developing intelligent system that support automated reasoning on UML class diagrams, so as to provide support during the specification phase of software development. We have already started experimenting such systems. In particular, we have represented UML diagrams in $\mathcal{DLR}$ and used DL reasoners, specifically

FACT [18] and RACER [16], for reasoning on UML class diagrams. Although such DL reasoners do not yet incorporate all features required by our formalization (e.g., support for identifiers), the first results are encouraging.

References

1. Sonia Bergamaschi and Bernhard Nebel. Acquisition and validation of complex object database schemata supporting multiple inheritance. *Applied Intelligence*, 4(2):185–203, 1994.
2. Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide*. Addison Wesley Publ. Co., Reading, Massachussetts, 1998.
3. Andrea Calì, Diego Calvanese, Giuseppe De Giacomo, and Maurizio Lenzerini. A formal framework for reasoning on UML class diagrams. Submitted for publication, 2001.
4. Diego Calvanese, Giuseppe De Giacomo, and Maurizio Lenzerini. On the decidability of query containment under constraints. In *Proc. of the 17th ACM SIGACT SIGMOD SIGART Symp. on Principles of Database Systems (PODS'98)*, pages 149–158, 1998.
5. Diego Calvanese, Giuseppe De Giacomo, and Maurizio Lenzerini. Reasoning in expressive description logics with fixpoints based on automata on infinite trees. In *Proc. of the 16th Int. Joint Conf. on Artificial Intelligence (IJCAI'99)*, pages 84–89, 1999.
6. Diego Calvanese, Giuseppe De Giacomo, and Maurizio Lenzerini. Identification constraints and functional dependencies in description logics. In *Proc. of the 17th Int. Joint Conf. on Artificial Intelligence (IJCAI 2001)*, 2001. To appear.
7. Diego Calvanese, Maurizio Lenzerini, and Daniele Nardi. Description logics for conceptual data modeling. In Jan Chomicki and Günter Saake, editors, *Logics for Databases and Information Systems*, pages 229–264. Kluwer Academic Publisher, 1998.
8. Diego Calvanese, Maurizio Lenzerini, and Daniele Nardi. Unifying class-based representation formalisms. *J. of Artificial Intelligence Research*, 11:199–240, 1999.
9. Tony Clark and Andy S. Evans. Foundations of the Unified Modeling Language. In David Duke and Andy Evans, editors, *Proc. of the 2nd Northern Formal Methods Workshop*. Springer-Verlag, 1997.
10. Francesco M. Donini, Maurizio Lenzerini, Daniele Nardi, and Andrea Schaerf. Reasoning in description logics. In Gerhard Brewka, editor, *Principles of Knowledge Representation*, Studies in Logic, Language and Information, pages 193–238. CSLI Publications, 1996.
11. Andy Evans, Robert France, Kevin Lano, and Bernhard Rumpe. The UML as a formal modeling notation. In Haim Kilov, Bernhard Rumpe, and Ian Simmonds, editors, *Proc. of the OOPSLA'97 Workshop on Object-oriented Behavioral Semantics*, pages 75–81. Technische Universität München, TUM-I9737, 1997.
12. Andy Evans, Robert France, Kevin Lano, and Bernhard Rumpe. Meta-modelling semantics of UML. In H. Kilov, editor, *Behavioural Specifications for Businesses and Systems*, chapter 2. Kluwer Academic Publisher, 1999.
13. Andy S. Evans. Reasoning with UML class diagrams. In *Second IEEE Workshop on Industrial Strength Formal Specification Techniques (WIFT'98)*. IEEE Computer Society Press, 1998.
14. Martin Fowler and Kendall Scott. *UML Distilled – Applying the Standard Object Modeling Laguage*. Addison Wesley Publ. Co., Reading, Massachussetts, 1997.
15. Volker Haarslev and Ralf Möller. High performance reasoning with very large knowledge bases: A practical case study. In *Proc. of the 17th Int. Joint Conf. on Artificial Intelligence (IJCAI 2001)*, 2001.
16. Volker Haarslev and Ralf Möller. RACER system description. In *Proc. of the Int. Joint Conf. on Automated Reasoning (IJCAR 2001)*, 2001.
17. David Harel and Bernhard Rumpe. Modeling languages: Syntax, semantics and all that stuff. Technical Report MCS00-16, The Weizmann Institute of Science, Rehovot, Israel, 2000.
18. Ian Horrocks. Using an expressive description logic: FaCT or fiction? In *Proc. of the 6th Int. Conf. on Principles of Knowledge Representation and Reasoning (KR'98)*, pages 636–647, 1998.
19. Ian Horrocks and Peter F. Patel-Schneider. Optimizing description logic subsumption. *J. of Logic and Computation*, 9(3):267–293, 1999.
20. Thomas Kirk, Alon Y. Levy, Yehoshua Sagiv, and Divesh Srivastava. The Information Manifold. In *Proceedings of the AAAI 1995 Spring Symp. on Information Gathering from Heterogeneous, Distributed Enviroments*, pages 85–91, 1995.

21. D. McGuinness and J. Wright. Conceptual modelling for configuration: A description logic-based approach. *Artificial Intelligence for Engineering Design, Analysis, and Manufacturing Journal*, 12:333–344, 1998.
22. Ulrike Sattler. *Terminological Knowledge Representation Systems in a Process Engineering Application*. PhD thesis, LuFG Theoretical Computer Science, RWTH-Aachen, 1998.

Development of Formally Verified Object-Oriented Systems with *Perfect Developer*

David Crocker

Escher Technologies Ltd.
3 Archipelago Business Park, Lyon Way, Frimley, Camberley GU16 5ER, UK
Web: `www.eschertech.com` Email: `dcrocker@eschertech.com`

Abstract. *Perfect Developer* (formerly known as the Escher Tool) is a highly productive system for developing reliable software systems using object-oriented methods. Dynamic binding and aliasing are carefully controlled to make the verification problem tractable.

1 Background

Formal methods have been used for many years in the development of safety-critical software but have yet to make it to mainstream software development. Barriers to wider use of formal methods include the requirement for users of formal methods tools to have extensive mathematical knowledge, the labour associated with assisting tools in discharging proof obligations, and the lack of support in most tools for object-oriented methods. These issues are addressed by *Perfect Developer*.

2 Limitations of existing O-O languages

The use of existing object-oriented program languages to develop formally verified software runs into both technical and practical difficulties.

The primary technical difficulties we have identified are:

- Unconstrained polymorphism. Variables, parameters and return values with non-primitive types are typically all polymorphic (i.e. an object of any class derived from the declared type is acceptable). This greatly increases the potential for dynamic binding and makes specifications and code very hard to reason about.
- Default reference semantics. The use of reference semantics as the default or only semantics for assignment and parameter passing of class variables greatly increases the potential for aliasing. When combined with polymorphism and dynamic binding, this leads to many situations in which it is impossible to prove that parameters to a class method (including the 'self' or 'this' parameter) are all distinct, which typically makes it impossible to prove correctness of methods that change their parameters or 'self'. We have observed that unintentional aliasing also makes a significant contribution towards software errors.

The practical difficulties are:

- Syntax for preconditions, postconditions, invariants and many other constructs needed for specification are not provided in the language. The usual solution is to insert these constructs as specially-formatted comments. This apparent demoting of specifications conveys the wrong message to developers.

- To make formal methods of software development acceptable to the mass market, the additional time spent writing specifications must be balanced by time savings elsewhere. The obvious solution is to generate code automatically from specifications; however, this also requires the language to be extended (by providing a syntax for an omitted code body).
- Programming languages do not support data refinement, which is a key tool in object-oriented software development using formal methods.

These difficulties mean that the most that can be achieved using existing programming languages is extended static checking (e.g. ESC/Java [1]). Such tools may be a useful bridge between existing software development practice and true verified software development; but even if the user can be persuaded to annotate the program with preconditions, invariants etc., complete formal verification is far from possible.

3 Perfect Developer

Perfect Developer (a development system for producing perfect software) takes the approach that the notation must resemble typical programming notations (i.e. avoid mathematical notation) but that code must take second place to specifications. Accordingly, it uses its own notation [2]. Concepts that can be expressed in the *Perfect* language include:

- Classes with single inheritance and dynamically bound methods
- Parametric polymorphism
- Class invariants and type constraints
- Method preconditions, postconditions, variants and post-assertions
- Quantification over sets, bags and sequences
- Expected behaviour of the system as a whole and of subsystems, including 'what if' scenarios

To avoid excessive use of unconstrained polymorphism, we distinguish between the type **T** and the type **from T**, where **T** is any non-final class (Ada 95 makes a similar distinction). Thus the user indicates explicitly where polymorphism is required.

In order to reduce the aliasing problem, *Perfect* uses value semantics by default for assignment and parameter passing. To avoid excessive copying, value semantics are simulated using reference semantics; copying is avoided wherever possible and where copying is necessary, typically only some part or parts of an object need to be copied. Reference types are provided for those situations in which intentional aliasing is required.

To develop a software system or component with *Perfect Developer*, a set of required properties of the system or component is specified. Hardware devices and other subsystems with which the software will interact may also be described and relevant behaviour specified. An assembly of classes is then designed to model and encapsulate the stored data and perform the required operations. Contracts are written for the class methods. *Perfect Developer* will attempt to generate code to satisfy the contracts where none has been provided. The developer may refine both class data and method code in order to meet performance targets.

Proof obligations are generated asserting that all contracts are honoured by both parties, that required properties will be observed, that all constructs will terminate and that refinements are valid. *Perfect Developer* attempts to discharge them using a fully automatic theorem prover, on the grounds that most software developers have neither the skill nor the time to assist in discharging proof obligations.

Final code is generated in C++ (code generation in Java and Ada 95 is under development).

4 Handling inheritance and dynamic binding

Dynamic binding continues to be a potential problem in that calls to dynamically bound methods cannot be expanded during verification.

If the software system is to be validated without regard to extensibility, it would be possible to enumerate the set of possible types of each polymorphic variable and perform validation with respect to all possible types of all polymorphic variables. We do not currently do this but may offer it as an option in the future.

A better solution is to recognise that in any family of classes with a common ancestor, for any method declared in the common ancestor and defined or redefined in each class, all the method declarations implement some common purpose. In some cases the method represents the definition of some property of the class; in other cases, the method modifies the class so as to make it satisfy some property. In the latter case, a non-compiled or 'ghost' dynamically bound function can be defined to express the property and the original method specified in terms of this function. Although this places a greater burden on the user, it helps greatly in clarifying the specification of dynamically bound methods as well as making them amenable to validation.

We note in passing that most class methods must be validated separately in each non-abstract class into which they are inherited without being overridden, since it is frequently the case that a method definition is valid in the class in which it is defined but is invalid in the context of a derived class (e.g. because it does not take account of additional variables in the derived class).

5 A large case study: *Perfect Developer* itself

Perfect Developer is itself implemented in *Perfect* apart from the user interface functions in the IDE module.

At the time of writing (May 2001), the source for the compiler/verifier comprises 105000 lines of *Perfect* (including comments) from which 176000 lines of C++ (without comments) are generated.

When validation of the entire system is performed, 115000 proof obligations are generated. Using default settings, the theorem prover discharges 87.6% of these in under 5 days. The success rate is currently increasing by between 1 and 2% per month as we improve the prover and eliminate specification and coding errors, while at the same time the average time spent on each obligation has decreased in the last few months from 5 seconds down to 3.5.

Analysis of unproven obligations indicates that about half are the result of incompletely specified contracts (due in part to the bootstrap process used to develop *Perfect Developer*) and most of the rest are provable in principle but beyond the capability of our present prover within a reasonable time limit. However, failed proof obligations do occasionally reveal incorrect coding or an inconsistent specification and did in one case reveal an error in the *Perfect* language definition itself.

6 Conclusions and future work

We have shown that formal methods can be used to develop a large and complex application in an object-oriented style with high productivity. Despite the relative immaturity of our prover, we have achieved a substantial degree of automated validation.

Perfect Developer is currently available in a teaching and evaluation edition. Commercial release is due in September, by which time we expect to support exceptions and multithreading in the language. Work continues on the theorem prover and we expect that the use of term indexing techniques and better unification algorithms will significantly improve the speed and success rate of validation.

References

1. David L. Detlefs, K. Rustan M. Leino, Greg Nelson, and James B. Saxe. Extended Static Checking. Research Report 159, Compaq Systems Research Center, December 1998. Available at http://research.compaq.com/SRC/esc/.
2. *Perfect* Language Reference Manual. Escher Technologies Ltd., March 2001.

Towards Verifiable Specifications of Object-oriented Frameworks
A Case Study

Jörg Meyer, Arnd Poetzsch-Heffter

FernUniversität Hagen

Abstract. The specification of object-oriented frameworks has to fulfill several requirements. It should document the behavior of the classes in an abstract, implementation independent way. It should be formally founded so that the correctness of implementations can be proved w.r.t. it. Last but not least, it should provide an appropriate basis for the verification of programs using the framework. In this paper, we present a case study that focuses on the first two requirements. It shows an abstract specification of a linked list implementation with shared objects, sketches the underlying formal framework, and explains the necessary proof steps.

1 Introduction

The specification of an object-oriented program framework has to fulfill several requirements. It should document the classes of the framework in an abstract, implementation independent way. Implementation independency is necessary to hide implementation aspects that should not be exploited by a user of the framework. Abstraction is needed to raise the level of specification. As a second requirement, a specification should be verifiable, i.e. a specification technique should be embedded into a formal setting and complemented with verification rules and proof techniques. Formalization helps to clarify the semantics of the specification language. Verification is needed to establish the correctness of code. As a third requirement, the specification of a module M should provide an appropriate basis to verify the correctness of a module N that uses M.

Most of the work that was done about specification and verification of OO-programs has focussed either on specification or on verification. In the area of specification, the main goals were the development of easy to use and expressible specification languages; the precise formal relationship between specifications and programs – a prerequisite of formal verification – was of minor interest. In the area of verification, the focus was on the formal proof rules and techniques. The remaining interesting challenge is to apply the verification techniques to prove the correctness not only of some isolated properties, but of complete interface specifications.

Contents. This paper investigates the relation between specification and verification. It presents the interesting parts of a case study, in which we verified the specification of a list implementation consisting of three classes (cf. [LMMPH00] for a complete report). The goal of the presentation is to describe important aspects and problems that have to be dealt with when verification techniques are applied to realistic specifications. In particular, we illustrate how abstraction can be handled and demonstrate the complexity that results from aliasing. To our knowledge, it is the first time that an OO-program with complex aliasing is proved correct. In Section 2, we present and discuss the specification. Section 3, sketches the proof technique.

Relation to Other Work. The work is related to interface specification techniques for OO-programs. In particular, the used technique builds on the two tiered Larch approach (cf. [GH93,Lea97]). That is we use a general specification language to express abstract properties – in our case the specification language of PVS – and an interface specification language to describe program interfaces. In this presentation, only some constructs of the interface specification language will be illustrated.

For verification, we use a Hoare logic (see [PHM99]) and an interactive program prover that communicates with PVS ([MPH00]). Recently a number of different approaches to the verification of Java-like OO-programs have been investigated. The LOOP group developed a translator that generates PVS theories from a given Java program ([JvdBH+98]). The theories capture the program behavior and can be used to verify program properties in PVS ([Hui00]). In his thesis, David von Oheimb formalized a Java subset and a corresponding Hoare logic in Isabelle. His work focuses on meta-theory, in particular type safety and correctness and completeness properties of the logic (see [vO01]). A dynamic logic for a Java subset is presented in [Bec00].

For brevity, we can't treat modularity properties in this extended abstract, although we consider them very important. The interested reader is refered to [Mül01] for this topic.

2 Specifying Object-oriented Programs

In this section, we describe the specification technique along with parts of an implementation for doubly linked lists. We start with the abstract interface specification. Then we show how specification and implementation are related.

2.1 The Specification of Class DList

A class specification consists of a list of invariants (keyword `inv`) and a list of method specifications. A method specification consists of an optional requires clause (keyword `req`) and a list of pre-post-pairs. The meaning of such a specification is given by desugaring it into Hoare triples. Each invariant has to be maintained by all public methods of the program[1]. The condition stated in the requires clause, if any, may be assumed in the prestate. In addition, each pre-post-pair constitutes a Hoare triple; again the requires clause, if any, is conjoined to the precondition. To keep things simple, we do not use more elaborate specification constructs like e.g. old-expressions.

The example we consider here is centered around the class `DList`. The specification is given along with the external visible parts of the class declaration. We present only those parts that are needed in the following. The syntax of formulas follows the syntax of the PVS language (see [COR+95]). Capital letters denote logical variables holding values of PVS builtin or user defined types. Logical variables are used to relate variable values in preconditions to those in postconditions. `result` is used as a special variable representing the value returned by a method:

[1] We use modularity techniques to decrease the number of methods for which the invariant has to be shown (see [Mül01]), but this is beyond the scope of this abstract.

```
class DList
{
 inv X: wfDList(X,$);

 public static DList empty()
 pre  TRUE;
 post ADList(result,$) = null;
 pre  alive(X,$) AND $=S;
 post ($==S)(X);

 public DList rest()
 req  ADList(this,$) /= null;
 pre  ADList(this,$) = L;
 post ADList(result,$) = cdr(L);
 pre  alive(X,$) AND $=S;
 post ($==S)(X);
```

```
 public int first()
 req  ADList(this,$) /= null;
 pre  ADList(this,$)=L;
 post aI(result) = car(L);
 pre  $=S; post $=S;

 public boolean isempty()
 pre  ADList(this,$) = L;
 post aB(result) = null?(L);
 pre  $=S; post $=S;

 public void app(int i)
 pre  ADList(this,$)=L AND
      I=aI(i) AND T=this;
 post ADList(T,$)
   = append(L,cons(I,null));
}
```

To express interface properties of OO-programs in an abstract way, three ingredients are necessary. (1.) We have to be able to refer to the object store. (2.) We need a functional vocabulary to express the behavior of the methods. (3.) Abstraction functions are necessary to relate the implementation to the abstract behavior.

To refer to the object store, we use the global program variable \$ of type `Store`. As we will show later, `Store` is an abstract data type. The predicate `alive` checks for an object whether it is allocated in a given store. As shown by the specification above, the methods `first` and `isempty` do not modify the object store. The methods `empty` and `rest` do not modify objects that are alive in the prestate; `(S1==S2)(X)` is a derived predicate on stores saying that every location (= instance field) reachable from object `X` holds the same value in stores `S1` and `S2`. The frame behavior of method `append` is not specified.

To express the functional method behavior in the example, we used the data type `list[int]` of PVS with constant `null`, functions `car`, `cdr`, `cons`, `append`, and predicate `null?`. In other examples, specific data types have to be designed to describe the abstract behavior of a class or framework. The relation between the abstract level and the implementation level is captured by abstraction functions and predicates. The predicate `wfDList` expresses the fact that the link structure of a list is well formed in a given store. The function `ADList` maps the given object and objects referenced by it in a given store to a PVS list. For technical reasons, we use abstraction functions as well for the basic data types boolean and int to map Java values to PVS values (`aB` and `aI`). For instance, the specification of `append` reads as follows: Abstracting the this-object in the poststate yields a list with new last element `I`. Again, we like to point out that the specification does not refer to any implementation detail.

2.2 The DList Implementation

In the following, we relate the above specification to the implementation of class `DList` and two auxiliary classes. The subsection focuses on those aspects that are needed to make an interface specification verifiable.

In the example, doubly linked lists are implemented by a `DList`-object as list header and a sequence of `NodeL`-objects as shown in Figure 1. The class `NodeL` is a subclass of a class `Node`. This separation into two classes was done to illustrate some aspects of inheritance.

```
class DList
{
 protected NodeL firstNode;
 protected NodeL lastNode;

 public static DList empty() {...}
 public DList rest() { ... }
 public int first() {
   Node f = this.firstNode;
   int k = f.getElem();
   return k;
 }
 public boolean isempty() { ... }
 public void app(int i) { ... }
}
```

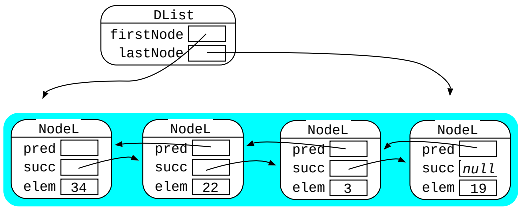

Fig. 1. Store Layout of a Doubly Linked List

Objects of class `Node` can be used in a very general way to create linked data structures, where each node holds a value of type `int`. The abstraction function `ANode` yields the `int`-value of the instance variable `elem`.

```
class Node
{
 protected int  elem;
 protected Node pred, succ;

 public int getElem()
 pre  $=S AND T=this;
 post aI(result) = ANode(T,S);
 pre  $=S; post $=S;

 public Node getPred()
 pre  $=S AND T=this;
 post result =
      S@@loc(T,Node?pred);
 pre  $=S;  post $=S;

 public Node getSucc()
 pre  $=S AND T=this;
 post result =
      S@@loc(T,Node?succ);
 pre  $=S;  post $=S;
}
```

The specification of method `getPred` shows how the object store can be accessed: `S@@loc(T,Node?pred)` denotes the value held by instance variable `Node?elem` of object `T` in store `S`. All functions and predicates concerning the object store are formalized in PVS. In particular, `Node?elem` is a constant. Type `Store` represents the abstract data type of object stores, `Location` is the set of instance variables, `Value` is a common type for all values and references occurring in the programming language. The following functions are provided: Let `l` be a location, `s` be a value of type store, `id` be a type identifier of a non-abstract class, and `v` be of type `Value`:

– `update(s,l,v)` returns the store, where `l` in `s` is updated with `v`.
– `new(s,id)` returns a reference to a fresh object allocated in `s`.
– `s##id` returns the store after allocating an object of type `id` in store `s`.

- `s@@l` returns the value stored in location `l` in store `s`.
- `alive(v,s)` returns *true* if the object (value) refered by `v` is allocated in `s`, *false* otherwise.

The semantics of the object store is specified by fourteen axioms. We present two of them to give an impression:

```
store1:  AXIOM L1 /= L2 => update(S, L1, X)@@L2 = S@@L2
store10: AXIOM NOT alive(new(S, T), S)
```

Axiom store1 describes that an update of some location does not affect other locations. Axiom store10 describes that some newly allocated object in a certain store was not alive in that store before allocation.

Based on class `Node`, we define a class `NodeL`. It contains an additional method and has a specification that guarantees the specific structure of the `NodeL`-objects implementing doubly linked lists (cf. the grey shaded area in Fig. 1).

```
class NodeL extends Node
inv  X: wfNodeL(X, $);
{
 public static NodeL initNodeL(int i)
 pre  I=aI(i);
 post ANodeL(result,$) = cons(I,null);
 pre  $=S;
 post result = new(S,NodeL) AND
      $=update(S##NodeL, loc(result,Node?elem), i);

 public int appback(NodeL n)
 req  n/=null AND n/=this AND lstNode?(this,$)
        AND fstNode?(n,$) AND lstNode?(n,$);
 pre  succn(X,$,N)=this AND ANodeL(X,$)=L AND ANodeL(n,$) = M;
 post ANodeL(X,$) = append(L,M);
 pre  $=S AND T=this AND X=n;
 post $=update(update(S, loc(T,Node?succ), X), loc(X,Node?pred), T);

 public NodeL getLast()
 pre  T=this AND $=S;
 post $=S AND result/=null AND
      (EXISTS (N:nat): succn(T,S,N)=result AND succn(T,S,N+1)=null);
}
```

To formalize the link structure for `NodeL`-objects, the following declarations and predicates are used. They should give an impression of what has to be available in a verification framework. For a detailed understanding, we assume that the reader is familiar with PVS syntax. By `<=` we denote the subtype relation on Java types which are represented by the constructor `ct` applied to their name:

```
NodeLObj?(X): bool = typeof(X) <= ct(NodeL)
NodeLObj?(X): TYPE = (NodeObj?)

fstNode?(X,S):bool = NodeLObj?(X) AND X/=null => S@@loc(X,Node?pred)=null
lstNode?(X,S):bool = NodeLObj?(X) AND X/=null => S@@loc(X,Node?succ)=null

predn((X: NodeObj), S, (n: nat)): RECURSIVE NodeLObj =
  IF X=null OR n=0 THEN X ELSE predn(S@@loc(X, Node?pred), S, n-1)
```

```
  ENDIF MEASURE n

succn((X: NodeObj), S, (n: nat)): RECURSIVE NodeLObj =
  IF X=null OR n=0 THEN X ELSE succn(S@@loc(X, Node?succ), S, n-1)
  ENDIF MEASURE n

wfNodeL((X: NodeLObj), S): bool =
  (EXISTS (i: nat): predn(X, S, i) = null)                    AND %(1)
  (EXISTS (k: nat): succn(X, S, k) = null)                    AND %(2)
  (NOT fstNode?(X,S) => typeof(S@@loc(X,Node?pred)) <= ct(NodeL) AND %(3)
   S@@loc(S@@loc(X,Node?pred), Node?succ) = X)                AND %(4)
  (NOT lstNode?(X,S) => typeof(S@@loc(X,Node?succ)) <= ct(NodeL) AND %(5)
   S@@loc(S@@loc(X,Node?succ), Node?pred) = X)                    %(6)
```

The predicate `fstNode?`(`lstNode?`) holds for a `NodeL`-object X, if X has no predecessor (successor). `predn`(`succn`) yields the n-th predecessor (successor) of a `NodeL`-object. This allows us to formulate a wellformed condition for X using the predicate `wfNodeL`: X is well formed, if there exists a predecessor(successor)-object, whose `pred`(`succ`)-location is `null` (1+2). This guarantees `NodeL`-structures to be non cyclic. (3+5) describe that if X has a predecessor(successor) Y then Y is of type `NodeL`. (4+6) guarantee that inner objects of a `NodeL`-structure are correctly linked, by reaching itself via its predecessor(successor).

Wellformedness of `NodeL`-structures is a prerequisite for a correct method execution and abstraction. The wellformed condition `wfNodeL` constitutes the invariant for objects of the `NodeL` class. As expressed by the invariant clause of the interface specification of class `NodeL`, each method has to preserve this invariant.

To specify the functional behavior of `NodeL`'s methods an abstraction is needed to capture functional list properties. Therefore we use the following abstraction function with signature `Value, Store -> list[int]`, which abstracts `NodeL`-structures to the generic PVS list data-type list with type parameter `int`:

```
ANodeLn((X: NodeObj), (S: Store), (n: nat)):
 RECURSIVE list[int] = IF X=null OR n=0 THEN null
 ELSE cons(aI(S@@loc(X,Node?elem)),
           ANodeLn(S@@loc(X,Node?succ), S, n-1))
 ENDIF MEASURE n
```

Class `NodeL` extends class `Node` and adds a method `public int appback (NodeL n)`, which concatenates a `NodeL`-structure and a `NodeL`-object, referred by `this` and `n`. By this method we demonstrate the implementation dependency of a specification in contrast to the specification of class `DList`. The implementation dependency enables verification but is therefore not suitable for documentation and reuse. A basic requirement for a successful execution is that `n` refers a single `NodeL`-object and that `this/=n` holds. Furthermore appending is only allowed at the last object of a `NodeL`-structure. These requirements are summarized in the req-clause of method `appback`:

```
req n/=null AND n/=this AND lstNode?(this,$)
            AND fstNode?(n,$) AND lstNode?(n,$);
```

Furthermore `appback` has (1) a functional and (2) an environment behavior specification. (1) guarantees that `appback` does in fact let `n` become the new successor of `this` and all possible tails of the `NodeL`-structure with `this` at the end stay unchanged. (2) specifies that the store is changed by two location updates.

The specification techniques shown above are as well applicable to Java's interface types. To specify interface types, i.e. types without implementations, abstraction techniques can be exploited. Another interesting specification aspect occurs together with inheritance. A class S inherits a method m from class T without overriding the implementation. Nevertheless, a refined specification can be needed in the subclass. In our example, class `NodeL` inherits method `getSucc` from class `Node` and refines its specification. In addition to the specification given above class `NodeL` requires from its implementation that `pre ANodeL(this,$) = L; post ANodeL(result,$) = cdr(L);` holds.

Abstraction Functions for Class DList. Equipped with the datatypes and specification primitives above, we now present the rest of the specification of class `DList`, i.e. the wellformed condition and the abstraction. Both are needed to verify the implementation of `DList`. The wellformed condition used in the invariant looks as follows. A list is considered to be empty, if `firstNode` refers `null`. The abstraction `ADList: [Value, Store -> list[int]]` reuses the abstraction of `NodeL` and is equal to the abstraction of the `NodeL` object referred by `firstNode`.

```
wfDList((X: DListRef), S): bool =
 S@@loc(X,DList?firstNode)=null OR
  S@@loc(X,DList?firstNode)/=null AND
  cS@@loc(X,DList?lastNode)/=null AND
   EXISTS (n:nat): succn(S@@loc(X,DList?firstNode),S,n)=
                   S@@loc(X,DList?lastNode)

ADList_ax: AXIOM DListRef?(X) =>
            ADList(X, S)=ANodeL(S@@loc(X,DList?firstNode),S)
```

3 A logical Framework for Proof Construction

Within this section we give an overview of the formal framework used for specification and verification. As shown in the specification we have to express program independent and program dependent properties. Using PVS allows us to use the following technique: Theories containing formalizations of type identifiers, attribute-identifiers, and lemmata containing the subtype hierarchy are generated for all used classes. Program independent theories are generic w.r.t. the generated theories. Both parts together provide the formal background for specification and verification.

The used verification technique is based on a Hoare logic for the programming language we use. Partial correctness of programs w.r.t. their specification is shown by translating interface specifications into Hoare triples and proving them using the programming logic. Remaining implications which arise from the use of strengthening or weakening rules are proved by using PVS.

Figure 2 shows the result of the Hoare logic proof of the method `first` of class `DList`, where the functional property of `first` is proven. The use of some of the Hoare rules is displayed (formula parts touched by rules are underlayed grey): The var-rule allows to replace logical variables by local program variables, if they do not occur on the left hand side of an assignment; the inv-rule allows to conjoin formulas F to the pre- and postcondition of a triple, if F does not contain program variables or the variable `$` for the object store; the ex-rule allows to add existential quantifiers for logical variables, which do not occur in the postcondition. A proof outline embeds the information of a proof tree into the program text, which allows a flat representation of the proof tree. It can be read as follows. At the line containing the invocation of `getElem` the invocation statement is instantiated with the functional specification triple of that method, i.e. formal parameters within the specification are replaced by actual parameters. Above the method invocation, the axiom for location reads is instantiated. f, which is assigned a new value, is replaced by the term reading the location `this.firstNode` in store `$` in the postcondition and used as new precondition. The use of Hoare logic rules is displayed using the horizontal lines. Arrows point to the antecedent triple. The triple of the consequence is displayed outside the line-brackets. Strengthening and weakening steps are simply denoted by $\Longrightarrow$.

In the proof outline example, the program proof part is complete. It remains to show the implications, marked by $\Longrightarrow$. The proof that the implications hold is obvious in this example. This results directly from expanding the definitions of the abstraction functions and the axioms of the store formalization. The example shows, that in addition to the specification part, abstraction plays an important role during verification. Because of this degree of complexity, a theorem prover supporting these data type mechanisms is indispensable.

The complete proof for the example used in the case study was constructed with the Jive environment. The Jive-prover combines an interactive program prover with the general purpose theorem prover PVS to perform the program independent proof tasks.

4 Conclusion

In this extended abstract we showed some hot spots of a case study concerning the specification and verification of object-oriented programs. We demonstrated, how properties of object oriented programs can be described in a program independent abstract way. This allows for (1.) precise documentation of object-oriented programs and frameworks for reuse, and (2.) specifications, which can be directly used to prove the specified program properties. We presented specifications of non trivial program properties for the used list example, which are difficult to express precisely with operational specification techniques. Furthermore we gave a short sketch of the proof techniques, which are implemented within the Jive proof environment. The Jive system was used to construct the complete program proof of the case study coupled with the PVS prover.

Acknowledgments Marcel Labeth contributed to this work by constructing the program proof with the Jive-System and proving all program independent proof obligations with PVS. We also thank Peter Müller for his important contributions to this work.

References

[Bec00] Bernhard Beckert. A dynamic logic for java card. In *Proc. 2nd ECOOP Workshop on Formal Techniques for Java Programs, Cannes, France*. TR 269, Fernuniversität Hagen, 2000. Available from `www.informatik.fernuni-hagen.de/pi5/publications.html`.

[COR+95] J. Crow, S. Owre, J. Rushby, N. Shankar, and M. Srivas. *A Tutorial Introduction to PVS*, April 1995.

[GH93] J. V. Guttag and J. J. Horning. *Larch: Languages and Tools for Formal Specification.* Texts and Monographs in Computer Science. Springer-Verlag, 1993.

[Hui00] M. Huisman. *Reasoning about Java programs in higher order logic Using PVS and Isabelle.* PhD thesis, University of Nijmegen, 2000.

[JvdBH+98] B. Jacobs, J. van den Berg, M. Huisman, M. van Berkum, U. Hensel, and H. Tews. Reasoning about Java classes. In *Proceedings of Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, 1998. Also available as TR CSI-R9812, University of Nijmegen.

[Lea97] G. T. Leavens. Larch/C++ reference manual. Available from `http://www.cs.iastate.edu/~leavens/larchc++manual/lcpp_toc.html`, July 1997.

[LMMPH00] M. Labeth, J. Meyer, P. Müller, and A. Poetzsch-Heffter. Formal Verification of a Doubly Linked list implementation: A case study using the JIVE system. Technical Report 270, FernUniversität Hagen, 2000.

[MPH00] J. Meyer and A. Poetzsch-Heffter. An architecture for interactive program provers. In S. Graf and M. Schwartzbach, editors, *TACAS00, Tools and Algorithms for the Construction and Analysis of Software*, volume 276 of *Lecture Notes in Computer Science*, pages 63–77, 2000.

[Mül01] P. Müller. *Modular Specification and Verification of Object-Oriented Programs.* PhD thesis, FernUniversität Hagen, 2001.

[PHM99] A. Poetzsch-Heffter and P. Müller. A programming logic for sequential Java. In D. Swierstra, editor, *ESOP '99*, LNCS 1576. Springer-Verlag, 1999.

[vO01] D. von Oheimb. *Analyzing Java in Isabelle/HOL: Formalization, Type Safety and Hoare Logic.* PhD thesis, Technische Universität München, 2001.

$$\{\, this \neq null \wedge ADList(this, \$) \neq null \wedge ADList(this, \$) = L \,\}$$

```
public int first() {
```

$$\{\, this \neq null \wedge ADList(this, \$) \neq null \wedge ADList(this, \$) = L \,\}$$

$$\Longrightarrow$$

$$\{\, this \neq null \wedge \$@@loc(this, DList?firstNode) \neq null \wedge ADList(this, \$) = L \,\}$$

$$\Longrightarrow$$

$$\left\{ \begin{array}{l} \exists S, T : S = \$ \wedge T = \$@@loc(this, DList?firstNode) \wedge this \neq null \wedge \\ \$@@loc(this, DList?firstNode) \neq null \wedge ADList(this, S) = L \wedge T \neq null \end{array} \right\}$$

↓–[ex-rule]

$$\left\{ \begin{array}{l} this \neq null \wedge \$@@loc(this, DList?firstNode) \neq null \wedge T \neq null \quad \wedge \\ \$ = S \wedge T = \$@@loc(this, DList?firstNode) \wedge ADList(this, S) = L \end{array} \right\}$$

$$\Longrightarrow$$

$$\left\{ \begin{array}{l} this \neq null \wedge \$@@loc(this, DList?firstNode) \neq null \wedge \$ = S \quad \wedge \\ T = \$@@loc(this, DList?firstNode) \quad \wedge \\ ADList(this, S) = L \wedge T = S@@loc(this, DList?firstNode) \wedge T \neq null \end{array} \right\}$$

↓–[var-rule]

$$\left\{ \begin{array}{l} this \neq null \wedge \$@@loc(this, DList?firstNode) \neq null \quad \wedge \\ \$ = S \wedge T = \$@@loc(this, DList?firstNode) \wedge T_1 \neq null \quad \wedge \\ ADList(T_1, S) = L \wedge T = S@@loc(T_1, DList?firstNode) \wedge T \neq null \end{array} \right\}$$

↓–[inv-rule]

$$\left\{ \begin{array}{l} this \neq null \wedge \$@@loc(this, DList?firstNode) \neq null \quad \wedge \\ \$ = S \wedge T = \$@@loc(this, DList?firstNode) \end{array} \right\}$$

```
Node f; f = this.firstNode;
```

$$\{\, f \neq null \wedge \$ = S \wedge T = f \,\}$$

```
int k; k = f.getElem();
```

$$\{\, aI(k) = ANode(T, S) \,\}$$

↑–[inv-rule]

$$\left\{ \begin{array}{l} aI(k) = ANode(T, S) \wedge T_1 \neq null \wedge ADList(T_1, S) = L \quad \wedge \\ T = S@@loc(T_1, DList?firstNode) \wedge T \neq null \end{array} \right\}$$

↑–[var-rule]

$$\left\{ \begin{array}{l} aI(k) = ANode(T, S) \wedge this \neq null \wedge ADList(this, S) = L \quad \wedge \\ T = S@@loc(this, DList?firstNode) \wedge T \neq null \end{array} \right\}$$

$$\Longrightarrow$$

$$\{\, aI(k) = car(L) \,\}$$

```
return k;
```

$$\{\, aI(result) = car(L) \,\}$$

↑–[ex-rule]

$$\{\, aI(result) = car(L) \,\}$$

```
}
```

$$\{\, aI(result) = car(L) \,\}$$

Fig. 2. Example for a Proof Outline of a Hoare-logic Proof

A Model Theoretic Semantics of OCL

Peter H. Schmitt

Universität Karlsruhe
Institute for Logic, Complexity and Deduction Systems
D-76128 Karlsruhe, Germany
i12www.ira.uka.de/~pschmitt

Abstract. This paper proposes a model theoretic semantics of the Object Constraint Language (OCL), observing the OMG standard as close as possible

1 Introduction

The Unified Modeling Language, UML, has gained widespread acceptance as a standard for modelign object-oriented systems. The Object Constraint Language, OCL, is a part of UML used to add constraints to UML diagrams that can cannot be expressed in the visual models. The available descriptions, the OMG standard document [8] and the book [16], fall short of giving a rigorous semantic, and even syntactic description of the language. Deficiencies have been pointed out e.g. in [11, 4, 5, 7]. The purpose of this paper is to give a systematic definition of syntax and semantics of the OCL.

One way to provide OCL with a precise semantics is via a translation into a known, well understood specification language. This approach has been pursued e.g. in [6, 3]. As an additional advantage of this approach the translated expressions may be used as input to existing tools. The disadvantage is that those not familiar with the target language will gain nothing.

We describe here a semantics of OCL in an informal yet mathematically rigorous way, making use of naive set theory only. We believe that this is a common ground for all specification languages, like Z, Abstract Machine Notation, CASL, Isabelle or HOL, to name just a few. Formalizing our semantics in any of these will be straightforward complicated only by the requirement to get around the restrictions imposed by the chosen framework.

This research is supported by DFG within the KeY project, see the web page `http://i12www.ira.uka.de/~key`.

2 The UML Context

OCL expressions only make sense with respect to a given UML model. For the time being OCL expressions may only be attached to class diagrams. Figure 1 shows more precisely how syntax and semantics of OCL depend on their counterparts in UML. From a given UML class diagram we read off a set of *model types* $S_{\mathcal{D}}$. These will in Subsection 3.1 be extended to obtain the set $S_{\mathcal{D}}^{OCL}$ of all OCL types. The class diagram also provides a subtype ordering $<_{\mathcal{D}}$ on $S_{\mathcal{D}}$, which is in Definition 6 again extended to a subtype relation on all OCL types. The vocabulary that is used to built up OCL expressions also comes in two parts: the symbols in $F_{\mathcal{D}}$ that arise from the diagram $\mathcal{D}$, and pre-defined OCL operation symbols. OCL constraints occur as additions to

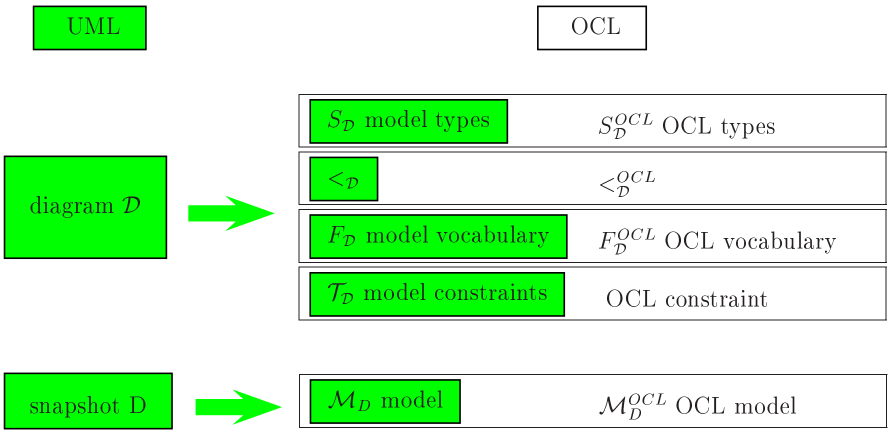

Fig. 1. Syntactic and semantic dependance of OCL on UML

UML diagrams in the form of invariants, preconditions and postconditions. But there is also information in $\mathcal{D}$ that goes beyond what is coded in $S_{\mathcal{D}}$, $<_{\mathcal{D}}$, and $F_{\mathcal{D}}$. A typical example are multiplicities of association ends. This information can easily be expressed by OCL expressions. For lack of space we do not pursue this here.

It is essential to distinguish between a class diagram $\mathcal{D}$ and a snapshot, or valid instance D of $\mathcal{D}$, see e.g. [13, pages 59–60] for a concise explanation. The class diagram completely determines the syntax of OCL-expressions over $\mathcal{D}$, while a snapshot D is needed to determine the meaning of an OCL expression. First a snapshot D gives rise to a many-sorted algebra $\mathcal{M}_D$ which will then be extended to an OCL-algebra $\mathcal{M}_D^{OCL}$ as depicted in the lower part of Figure 1. For the rest of the paper we assume that $\mathcal{D}$ is a fixed class diagram and D a valid instance of it.

2.1 The vocabulary of a UML diagram

The signature $\Sigma_{\mathcal{D}}$ consists of the set $S_{\mathcal{D}}$ of sorts, the set $F_{\mathcal{D}}$ of functions, and a subsort relation $<_{S,\mathcal{D}}$ on $S_{\mathcal{D}}$ as detailed in the following definitions.

Definition 1 $(S_{\mathcal{D}}, <_{\mathcal{D}})$**.**

1. *The set $S_{\mathcal{D}}$ of model types consists of all class symbols in $\mathcal{D}$.*
2. *The subtype relation $S_1 <_{\mathcal{D}} S_2$ holds if and only if type S_1 is declared a subtype of S_2 in $\mathcal{D}$.*
3. *$\ll_{\mathcal{D}}$ denotes the transitive, reflexive closure of $<_{\mathcal{D}}$.*

In particular, $S_{\mathcal{D}}$ contains symbols for all classes with the stereotype $\ll$ *enumeration* $\gg$ that happen to occur in $\mathcal{D}$.

The functions in $F_{\mathcal{D}}$ arise from various sources in $\mathcal{D}$ as detailed in the following definition.

Definition 2 $(F_{\mathcal{D}})$**.**

1. *For every association r in $\mathcal{D}$ and every two different association ends e_1, e_2 of r there is a function symbol $f_{r,e_1,e_2} \in F_D$.*
   If e_i is attached to the class S_i for $i = 1, 2$ then the function symbol receives the

corresponding signature: $f_{r,e_1,e_2} : S_1 \rightarrow Set(S_2)$.
In case the multiplicity at the end e_2 *is* 1 *the signature is:* $f_{r,e_1,e_2} : S_1 \rightarrow S_2$.
If the e_2*-end has stereotype* $\ll ordered \gg$ *then the signature is:*
$f_{r,e_1,e_2} : S_1 \rightarrow Sequence(S_2)$.

2. *For every attribute* a *of a class* S *in* $\mathcal{D}$ *there is a function symbol* $f_a \in F_{\mathcal{D}}$. *If* S_r *is the value type of* a *specified in* $\mathcal{D}$ *then* f_a *is given the signature* $f_a : S \rightarrow S_r$. *If* a *is a class attribute, (sometimes this is also called a static attribute), then* f_a *is a constant symbol of type* S_r.
3. *For every operation* c *of a class* S *with parameters of type* $S_1, \ldots, S_k$ *and result type* S' *there is a function symbol* $f_c \in F_{\mathcal{D}}$ *with signature* $f_c : S \times S_1 \times \ldots \times S_k \rightarrow S'$.
We will require that c *has no side effects, i.e.* c *satisfies the property* `isQuery()` *(see [9, p.2-25]).*
4. *For every association class* C *attached to an association* r*, where* r *associates the classes* S_1 *and* S_2 *there are symbols for the unary projection functions* pr_{S_1} *with signature* $C \rightarrow S_1$*, and* pr_{S_2} *with signature* $C \rightarrow S_2$.

We will restrict attention to binary relations. More than binary associations are rare anyway. The extension to associations with m association ends can be easily obtained by introducing m unary functions.

It is not clear if the e_2-end is allowed to be of multiplicity 1 **and** of stereotype $\ll ordered \gg$. If this case is possible we would suggest that the multiplicity takes precedence and the function have signature $f_{r,e_1,e_2} : S_1 \rightarrow S_2$ in clause 1 of Definition 2.

The explanations in [13, page 166ff] allow to attach a multiplicity different from 1 to attributes, including the exceptional case of multiplicity 0. This seems to have not been widely accepted. In [2] e.g. this possibility is not even mentioned. We will thus consider for each attribute a the associated function f_a to be total and single-valued.

From the point of view of abstract syntax the names of the symbols in $\Sigma_{\mathcal{D}}$ are irrelevant. But for all practical purposes it helps to stick to the following naming conventions:

Definition 3 (Naming Conventions).

1. *Sorts in* $S_{\mathcal{D}}$ *will be given the same name as the corresponding class. Sort names begin with an uppercase letter.*
2. *The function symbol* f_{r,e_1,e_2} *will be referred to by the role name of* r *at the association end* e_2. *If no role name is given, the name of the class attached to* e_2 *will be used. Function names start with a lowercase letter.*
3. *Function symbols arising from attributes or operations will carry the name of the attribute or operation. If attr is a static attribute of class* C *then the concrete syntax of the constant* f_{attr} *will be* $C.attr$. *This is in accordance with common usage, see e.g. [16, Section 3.5.3]. Here the dot is not used to indicate application of a property, but is simply part of a name.*
In Java it is possible to apply a static attribute of a class C *to instances of class* C. *If we wanted to allow the same behaviour on the OCL level we would have to introduce a static attribute attr in addition to the constant* $C.attr$ *a unary function symbol* $f_{attr} : C \rightarrow S$*, where* S *is the value type of attr.*

4. *The projection functions pr_{S_1} and pr_{S_2} for an association class C are considered as implicit attributes of C and denoted by lowercase class names s_1 and s_2. In case we want to also include n-ary associations we would, of course, have n projection functions.*

2.2 Semantics of UML diagrams

For any snapshot D of the UML class diagram $\mathcal{D}$ we will now define an associated many-sorted algebra $\mathcal{M}_D = (M_D, I_D)$ of signature $\Sigma_{\mathcal{D}}$. For ease of reading we will write $\mathcal{M}$ instead of $\mathcal{M}_D$ and I instead of I_D when there is no danger of confusion.

$\mathcal{M}$ may be viewed as a second-order algebra since function values will not always be elements of the universe of $\mathcal{M}$, but sometimes sets of elements. $\mathcal{M}$ is only the first step towards the OCL model $\mathcal{M}_D^{OCL}$ to be given in Section 4.

Definition 4 **($\mathcal{M}_D$).**

1. *For each sort symbol $S \in S_{\mathcal{D}}$ the domain $I(S)$ consists of all objects in the class S of the snapshot D plus one new element $\perp$ reserved to stand for the value of otherwise undefined terms. The universe M_D of $\mathcal{M}_D$ is the union of all type universes $I(S)$.*
2. *If S is a sort symbol arising from an association class attached to an association r between the classes S_1 and S_2, then $I(S)$ is the cartesian product of $I(S_1)$ and $I(S_2)$.*
3. *For sort symbols $S_1, S_2 \in S_{\mathcal{D}}$ with $S_1 \ll_{\mathcal{D}} S_2$ we stipulate $I(S_1) \subseteq I(S_2)$. For sort symbols $S_1, S_2 \in S_{\mathcal{D}}$ satisfying neither $S_1 \ll_{\mathcal{D}} S_2$ nor $S_2 \ll_{\mathcal{D}} S_1$ the sort universes $I(S_1)$, $I(S_2)$ are disjoint.*
4. *The function symbol $f_{r,e_1,e_2} \in F_{\mathcal{D}}$ with signature $S_1 \to Set(S_2)$ will be interpreted by the function $I(f_{r,e_1,e_2})$. For an arbitrary object $a \in I(S_1)$ $I(f_{r,e_1,e_2})(a)$ will be the set of all objects b in $I(S_2)$ that are in D linked to a via association r.*
5. *For function symbols $f_a : S \to S_r$ arising from an attribute a, the function value $I(f_a)(b)$ is the value of the attribute a of the object b of type S as given by the snapshot D.*
6. *For function symbols $f_c \in F_{\mathcal{D}}$ arising from query operations, the interpretation $I(f_c)$ is defined analogously to the previous clause. Query operations are not explicitly required to terminate. If operation c on the object b does not terminate then we set $I(f_c)(b) = \perp$.*
7. *The value of $I(f)$ for argument tuples containing one entry outside the required sort or one entry equal to $\perp$ equals $\perp$.*

Comments

1. If in clause 1 of the previous definition S is an abstract class, then S has no instances in D. In this case $I(S)$ is the set-theoretic union $I(S_1) \cup \ldots \cup I(S_k)$ where $S_1, \ldots, S_k$ are all immediate subclasses of S. This contradicts the position taken in [16, 4.3.1], which would imply $I(S) = \emptyset$ for an abstract class S. Our position is, however, in accordance with the semantics of an abstract class in [13, p. 117]:

   An abstract class may not have direct instances. It may have indirect instances through its concrete descendents.

2. If a is in particular a static attribute of class C then $I(C.a)$ is interpreted as an element of $I(T)$, where T is the type of a. If also a unary function f_a was introduced for the static attribute (see Definition 3 item 3), then we require in addition $I(C.a) = I(f_a)(o)$ for every $o \in I(C)$.
3. The approach just outlined treats association classes as classes, i.e. as sets of objects. In the above semantics an association class is **not** an association. Note that this does not contradict the UML metamodel. It is stated in [9, pages 2-20/2-21] that

   > AssociationClass is a subclass of both Association and Class (i.e., each AssociationClass is both an Association and a Class); therefore, an AssociationClass has both AssociationEnds and Features.

   The meta-model describes how to build syntactically correct diagrams, and gives restrictions on which model elements may be combined in which way. It has no effect on the semantics of diagrams.

3 The Syntax of OCL

The grammar for OCL in [8] is hard to understand. It is our goal to give here a human-oriented definition. We make use of the papers [12] which contains a description of the OCL syntax at the level of the UML metamodel, and [11] which gives a first account of a semantics for OCL.

We will present the syntax of OCL in two steps. In the first part the OCL type system will be explained. The second part contains a human-oriented description of the OCL grammar.

3.1 The OCL type system

Syntax of Type Expressions

Definition 5 (Type expressions). *Let $\mathcal{D}$ be a fixed class diagram. The type expressions with respect to $\mathcal{D}$ are as follows:*

1. *Integer, Real, Boolean, String*
   are type expressions. These are refered to as the simple OCL type expressions.
2. *OclType, OclAny, OclExpression, OclState*
   are type expressions. We call these meta type expressions.
3. *Any $s \in S_{\mathcal{D}}$, i.e. any class occuring in $\mathcal{D}$ is a type expression.*
   Following [16] we call these model types.
4. *If T is a type expression that is not itself of the form $Collection(T')$, $Set(T')$, $Bag(T')$, or $Sequence(T')$, then*

   $Collection(T)$, $Set(T)$, $Bag(T)$, $Sequence(T)$

   are type expressions. The types denoted by these expressions are usually refered to as collection types.

We will use $\mathrm{TE}_{\mathcal{D}}$*, or simply* TE *if no confusion is possible, to refer to the set of type expressions with respect to $\mathcal{D}$. In accordance with [10, Section 7.8.1] we refer to the types from clauses 1 and 2 as* basic types.

Comments

1. We consider the sentence "Collection, Set, Bag and Sequence are basic types as well." on [10, Page 7-7] contradicting the above stipulation of basic types as a plunder that will be remedied in a future release.
2. Unlike [12], we adhere to the standard and do not allow nesting of collection type constructors, i.e. $Set(Set(Integer))$ is not a legal type expression.
3. The accepted issue #3143 of the UML RTF proposes to drop the special OCL syntax for enumerations and use instead UML classes with stereotype $\ll$ *enumeration* $\gg$. This also makes the OCL type *Enumeration* superfluous, so we did omit it. Following [1] one might reintroduce it as the common supertype of all model types with stereotype $\ll$ *enumeration* $\gg$. But this remains to be decided.
4. $Set(OclType)$, $Set(OclAny)$, $Set(OclExpression)$, $Set(OclState)$ are legal type expressions.

Definition 6 (Direct Subtypes). *For type expressions $T_1, T_2 \in \mathrm{TE}_{\mathcal{D}}$ the subtype relation $T_1 <^{OCL}_{\mathcal{D}} T_2$ is the least relation satisfying the following conditions:*

1. *If T_1, T_2 are model types and T_1 is a subtype of T_2 in the UML model, i.e. $T_1 <_{\mathcal{D}} T_2$, then $T_1 <^{OCL}_{\mathcal{D}} T_2$.*
2. *$Integer \ <^{OCL}_{\mathcal{D}} \ Real$.*
3. *For all type expressions T, not denoting a collection type,*
   (a) *$Set(T) <^{OCL}_{\mathcal{D}} Collection(T)$*
   (b) *$Bag(T) <^{OCL}_{\mathcal{D}} Collection(T)$*
   (c) *$Sequence(T) <^{OCL}_{\mathcal{D}} Collection(T)$*
4. *If T is a model type or a basic type different from $OCLAny$, then $T <^{OCL}_{\mathcal{D}} \ OCLAny$.*
5. *If $T_1 \ <^{OCL}_{\mathcal{D}} \ T_2$ and C is any of the type constructors Collection, Set, Bag, Sequence, then $C(T_1) <^{OCL}_{\mathcal{D}} C(T_2)$.*

Definition 7 (Type conformance). *The transitive, reflexive closure of the subtype relation $<^{OCL}_{\mathcal{D}}$ is denoted by $\ll^{OCL}_{\mathcal{D}}$. If $T_1 \ll^{OCL}_{\mathcal{D}} T_2$ holds, we say that T_1 conforms to T_2.*

Semantics of Types The constructions explained in this subsection are performed with respect to a fixed snapshot D of a UML class diagram $\mathcal{D}$. As described in Section 2 we associate with D a many-sorted structure $\mathcal{M}_D = (M_D, I)$ consisting of the universe M_D and the interpretation function I. Let $S_{\mathcal{D}}$ denote the model types in $\mathcal{D}$, see Definition 1. Then every $C \in S_{\mathcal{D}}$ is interpreted via I as a subset of the universe M_D, i.e. $I(C) \subset M_D$.

The objective is to define the structure $\mathcal{M}^{OCL}_D = (M^{OCL}_D, I^{OCL})$ extending $\mathcal{M}_D$ such that for every OCL expression E the interpretation $I^{OCL}(E)$ is defined in $\mathcal{M}^{OCL}_D$. In this subsection we define $\mathcal{M}^{OCL}_D$ and the interpretation I^{OCL} on all type expressions. The definition of I^{OCL} will then be continued in Section 4.

When no confusion is possible we will supress the superscript OCL.

The universe of $\mathcal{M}^{OCL}_D$ is the union of all sort universes. Sort universes correspond to certain types. With the type *Integer*, for example, there is a sort universe $I(Integer)$ that consists of all integers plus the additional symbol $\perp$. For each model sort C there is the sort universe $I(C)$, as already explained in Definition 4, Clause 1. Remember that the special symbol $\perp$ for undefinied values is also an element of $I(C)$. In addition there are sort universes for $Set(C)$, $Bag(C)$ and $Sequence(C)$ and the universe for $Collection(C)$ will be the disjoint union of them. The sort universes for $Set(C)$, $Bag(C)$ and $Sequence(C)$ consist of abstract objects o, representing subsets, bags (multisets) or sequences of the sort universe of C respectively. The sort universe for *OclType* will consist of objects that are in one-to-one correspondance with the set consisting of model types and all basic OCL types, except *OclType*.

For example, for the pre-defined type *Integer* there will be an object $o_{Integer}$ in $M(OclType)$. In [10, Subsection 8.8.1.1] we read the somewhat mysterious sentence "All types defined in a UML model, or pre-defined within OCL, have a type." In our setting this now makes perfect sense: For every type S there is an object o_S in $M(OclType)$. We will refer to o_S as the *type object* of S.

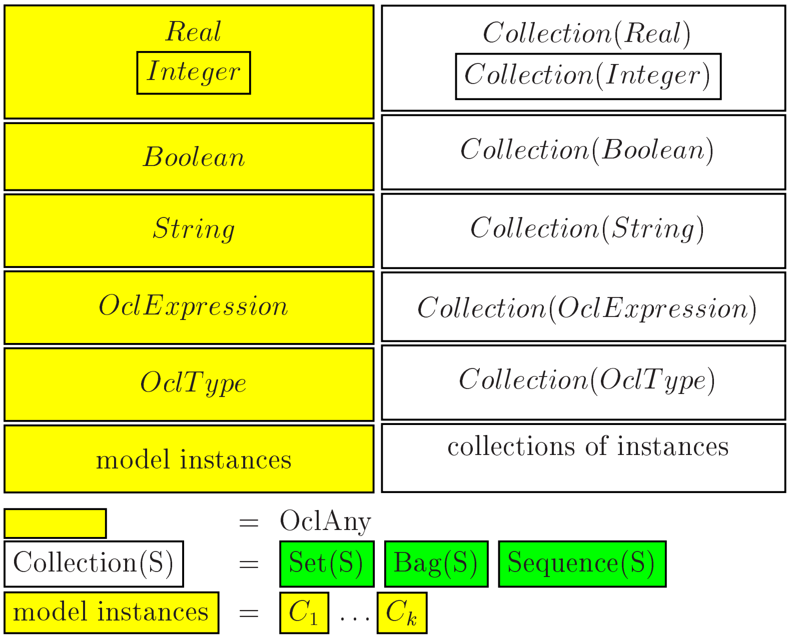

Fig. 2. Sort universes of $\mathcal{M}_D^{OCL}$

Figure 2 shows the sort universes of $\mathcal{M}_D^{OCL}$.

Definition 8. *1. For the OCL type expressions $S \in \{Integer, Real, Boolean, String\}$ the obvious universes $M'(S)$ are the integers, reals, booleans and all strings over a fixed alphabet A, respectively.*
$I(S) = M(S) = M'(S) \cup \{\perp\}$
Here $\perp$ is a new element denoting an undefined value. The sets $M'(S)$ are considered disjoint with the exception of $M'(Integer)$ being a subset of $M'(Real)$.

2. – *$I(OclType)$ = a set of objects in one-to-one correspondance with all basic or model types with the exception of OclType itself.*
– *$I(OclAny) = \bigcup\{I(S) \mid S$ a basic or model type$\}$*
– *$I(OclExpression) = \mathrm{TE}_{\mathcal{D}}$*

   – *$I(OclState)$ is not treated here.*
3. *for $S \in S_{\mathcal{D}}$ $I^{OCL}(S) = I(S) \cup \{\perp\}$, see Definition 4, Clause 1.*
4. – *$I(Set(T)) =$ set of all subsets of $I(T)$,*
   – *$I(Bag(T)) =$ set of all multisets of elements from $I(T)$,*
   – *$I(Sequence(T)) =$ set of all sequences of elements from $I(T)$,*
   – *$I(Collection(T)) = I(Set(T)) \cup I(Bag(T)) \cup I(Sequence(T))$*

The universe of the structure $\mathcal{M}_D^{OCL}$ is now taken to be the union of all $I(S)$.

Comments

1. While the standard [8] is precise on the meaning of *OclAny* the intended meaning of *OclType* is less clear. The definition adopted here seems a reasonable extrapolation. A more fundamental change which raises *OclType* to a class in the metamodel is proposed in [1].
2. We did not introduce type objects for collection types. The standard seems not to exclude this. But consider the following difficulty: Let o be the type object for type $Set(Integer)$ what is $o.allInstances$ supposed to be?
3. It would probably not harm to have a type object for *OclType* itself, but it will on the other hand not be of much help and certainly hard to swallow.

3.2 The Syntax of OCL Constraints

A constraints starts with a *header* fixing the context in which it is to be understood. Headers come in two forms, one for the classifier context, and one for the operator context. We start with the classifier context:

context (c **:**)? typeName **inv** expressionName? **:** OclExpression

The trailing question mark ? indicates optional elements; OCL keywords are set in **boldface**. 'typeName' could for example be the name of a class in the fixed UML diagram. In general we allow all type expressions that are not collection expressions as context type. It is possible to introduce a name for easy referencing of expressions. The optional parameter c will act very much like a variable of the type given by typename in the following OCL expression. Variable is here to be understood in the way it is used in formal logic. A header may define more than one expression:

context (c **:**)? typeName
  inv expressionName$_1$? **:** OclExpression$_1$
  ...
  ...
  inv expressionName$_n$? **:** OclExpression$_n$

Constraints for an operator context look like this:

context (c **:**)? typeName **::**opName**(**p_1**:** type$_1$**;** ... **;**p_k**:** type$_k$ **):**rtype
  {**pre** ,**post** } expressionName? **:** OclExpression

Here opName is meant to be the name of an operator defined on the given type. The list of parameters $p_1 \dots p_k$ may be empty and the return type, rtype, may be missing or both. As above, an operator constraint may contain more than one expression.

In the headers just shown OCL expressions have to be of type Boolean. Also the stereotype **inv** can only appear in a classifier context while the stereotypes **pre** and **post** can only show up in operator contexts.

We have added the optional parameter(c **:**)? also in the operator context to set it on equal footing with classifier contexts, though we have seen no example of this in the literature.

The definition of OCL expressions to be given presently depends on a fixed class diagram $\mathcal{D}$. More precisely, $\mathcal{D}$ uniquely determines the set $S_{\mathcal{D}}$ of model types and the set $F_{\mathcal{D}}$ of functions together with their sorting signature. Model types and type expressions for OCL have been introduced in Subsection 3.1 already.

Definition 9 (OCL function symbols). *The set $F_{\mathcal{D}}^{OCL}$ of function symbols admitted in forming OCL expressions for diagram $\mathcal{D}$ is*

1. *$F_{\mathcal{D}}$ (see Definition 2) plus*
2. *For any basic or model type S there is a constant symbol S in F with type$(S) =$ OclType,*
3. *All properties of the pre-defined OCL types as detailed in the standard [8, Section 7.8] plus*
4. *The constant symbol* result*. This is only needed for expressions within the context of an operator opname. The type of* result *will then be the return type of opname.*

Definition 10 (OCL expressions). *The set OCLExp of OCL expressions is the smallest set satisfying the following recursive conditions. At the same time we define for every OCL expression e its unique type type(e).*

1. *For every model type $t \in S_{\mathcal{D}}$ there is an unlimited number of variables v_t^i. Each variable v_t^i is in OCLExp with type$(v_t^i) = t$. The parameters of an operator constraint are special instances of variables of the type specified in their declaration.*
2. **self** *is a special variable, where type*(**self**) *is given by context information.*
3. **result** *is in OCLExp.*
   This is only allowed if the expression occurs in the context of an operator m in the stereotype **post**. *Then type*(**result**) $=$ *return type of m.*
4. *There are constant symbols for integers, reals, and strings, for example* 15, 7.88, *'Peter'. The precise syntax for these constants is given by the OCL grammar, where they are called* literals. *In addition there are constants for the two Boolean values, $true$ and $false$.*
5. *If f is a function symbol in F with argument types $t_1, \dots, t_k$ and result type t_r and $e_1, \dots, e_k$ are OCL expressions with type$(e_i) \ll_{\mathcal{D}}^{OCL} t_i$ for all $1 \le i \le k$ (see Definition 7) then*
   $f(e_1, \dots, e_k) \in OCLExp$ with type$(f(e_1, \dots, e_k)) = t_r$.
6. *If f is a function symbol in F with argument types $t_1, \dots, t_k$ and result type t_r where f is not the name of an operation and $e_1, \dots, e_k$ are OCL expressions with type$(e_i) \ll_{\mathcal{D}}^{OCL} t_i$ for all $1 \le i \le k$ (see Definition 7) then*

$f@pre(e_1, \ldots, e_k) \in OCLExp$ with $type(f(e_1, \ldots, e_k)) = t_r$.
OCL expressions containing @pre may only occur under the **post** *stereotype.*

7. *If e_1, e_2 are OCL expression of the same type, then $e_1 = e_2$, $e_1 <> e_2$ are OCL expressions of type Boolean.*
8. *If e is an OCL expression of type Boolean, e_1, e_2 are expressions of type t_1 and t_2 respectively and either $t_1 \ll t_2$ or $t_1 \gg t_2$ then*
   if e **then** e_1 **else** e_2
   is an OCL expression.
   $$type(\mathbf{if}\ e\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2) = \begin{cases} t_2 \text{ if } t_1 \ll t_2 \\ t_1 \text{ if } t_1 \gg t_2 \end{cases}$$

If need arises we will write more precisely $OCLExp_{\mathcal{D}}$ instead of $OCLExp$.

Comments

1. We allow the shorthand notation for **collect** as stated in the standard [10, Section 7.6.2] and also in [16, Subsection 3.6.11].
   Assume e is an OCL expression with type $Set(T)$, and $attr$ is an attribute of type T. Then $e.attr$ is a shorthand for $e \rightarrow \mathbf{collect}(c \mid c.attr)$. The same applies to functions arising from associations. Consider as an example the UML class diagram in Figure 3.

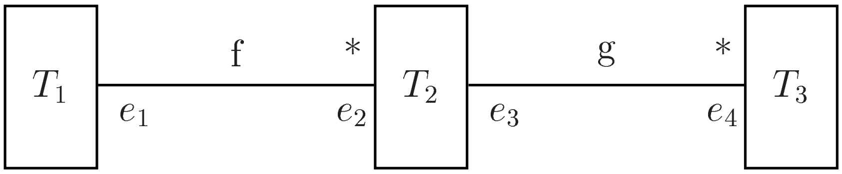

Fig. 3. Composition of navigation

   The OCL expression $\mathbf{self}.e_2$ in the context T_1 has type $Set(T_2)$. The expression $\mathbf{self}.e_2.e_4$ in the same context is short for $\mathbf{self}.e_2 \rightarrow \mathbf{collect}(c \mid c.e_4)$ and has type $Bag(T_3)$.
2. We have in clause 4 of the previous definition blurred the distinction between syntax and semantics. Adhering strictly to this distinction we should have introduced a constant symbol c_a for every integer a with the semantic interpretation $I(c_a) = a$. This seemed too much trouble for too little reason.
3. In Definition 10 we have used the functional notation as concrete syntax. This does not imply that we consider it superior to the concrete navigation syntax used in the OCL standard. We will in fact feel free to use both. Some remarks on the comparision of OCL notation to the notation used in the modeling language Alloy may be found in [15].

function syntax	navigation syntax
$g(f(c))$	$c.f.g$
$oclIsKindOf(self, Class)$	$self.oclIsKindOf(Class)$
$forAll(c, x, e)$	$c \rightarrow forAll(x \mid e)$

Definition 11 (Free variables). *For an OCL expression e we denote by* $\mathrm{FV}(e)$ *the set of free variables in e. The expression* **self** *is counted as a free variable.*

We trust that the notion of free variable, and of its counterpart bound variable, are intuitively clear. Here is an example:

$\mathrm{FV}(\mathbf{iterate}(e, v_t : t, v_{t_1} : t_1 = e_1 \mid f)) = \mathrm{FV}(e) \cup (\mathrm{FV}(f) \setminus \{v_t, v_{t_1}\})$

Definition 12 (OCL expressions in context).

1. *An OCL expression e may occur in the context*

   context *(c* **:***)? typeName* **inv** *expressionName?* **:** *e*

   only if

   (a) the postfix @pre and **result** *do not occur in e*
   (b) $\mathrm{FV}(e) = \{\mathbf{self}\}$ *if the parameter c is not specified. If c is specified, then we insist that* **self** *does not occur in e and* $\mathrm{FV}(e) = \{c\}$.
   (c) $type(e) = Boolean$.
2. *An OCL expression e may occur in the context*

   context *(c* **:***)? typeName*
   :: *opname*(p_1: *type*$_1$; ... ;p_k: *type*$_k$):*rtype*
   {**pre** , **post** } *expressionName?* **:** *e*

   only if

   (a) $\mathrm{FV}(e) = \{p_1, \ldots, p_k\} \cup \{\mathbf{self}\}$ *if c is not specified, and* $\mathrm{FV}(e) = \{p_1, \ldots, p_k\} \cup \{c\}$ *if c is specified.*
   (b) in the case of the **pre** *stereotype, @pre and* **result** *do not occur in e.*
   (c) $type(e) = Boolean$.

Comments and examples

1. OCL constraints have so far in the literature been considered with respect to a fixed context class, say S. Inspecting Definition 10 reveals that this information is only needed in determining the type of **self**. The same observation applies to [11, Definition 5]. Replacing **self** by the optional parameter $c : S$ specified in the context header we arrive at a definition of *OCLExp* not depending on context information.
   The notion of context is helpful, so we introduce it in Definition 12, thus arriving at a clear separation of concerns.
   The approach adopted here also solves another problem that has been kept under the rug in previous publications. Assume that e is an OCL expression in context c with $type(e) = \mathbf{Sequence}(t)$ and we form the new expression

   $$e \to \mathbf{iterate}(v_t : t, v_{t_1} : t_1 = e_1 \mid f)$$

   What should be the context of f? In most published examples it is not c, sometimes it is t, sometimes it is t_1.
   The above definitions at least give a clear answer, which we hope will also prove useful and in keeping with the spirit of OCL.

2. OCL allows as a shortcut to omit **self**. We assume in Definition 12 that this, and all similar shortcuts, have been removed.
3. Here are some examples for OCL expressions according to clause 5.
   If $f = +$ and $type(e_1) = type(e_2) = Integer$ then $e_1 + e_2$ is an expression of type $Integer$.
   There are two ways to derive the expression $e_1 + e_2$ within the OCL grammar in the standard [9]: first as a built-in operation on the type $Integer$ and second via the grammer rule for `additiveExpression`. This shows that one could simplify the grammar.
   If T is a class in $\mathcal{D}$, e an expression of type T and f an attribute for the class T with value type T_1 then $f(e)$ is in $OCLExp$ with $type(f(e)) = T_1$. In concrete OCL syntax this expression would be written as $e.f$.
   Let $f = f_{r,b_1,b_2}$ (see Definition 2) be a function symbol associated with the association ends b_1 and b_2 of an association r in $\mathcal{D}$. Assume that b_i is attached to class t_i and e is an OCL expression with $type(e) = T_1$. Then $f(e)$ is an OCL expression with $type(f(e)) = T_2$ if b_2 has multiplicity 1 and $type(f(e)) = Set(t_2)$ otherwise. In concrete OCL syntax this expression would be written as $e.b_2$ or $e.t_2$.
   If e is an expression with $type(e) = Set(T)$ then $size(e)$ is an OCL expression of type $Integer$, because $size$ is a built-in operator. In concrete syntax: $e \to size$.
4. If the type of e_1, e_2 in clause 7 is neither a meta type nor a collection type, then equality and inequality are built-in operations of the type $OCLAny$. Because of the restriction of the subtype relation in Definition 6 this does not cover equality between collection and meta types. Thus we have added clause 7 here.
5. Let e be an OCL expression of type **Sequence**(T), let v_T, v_{T_1} be variables of type T and T_1 respectively, e_1 and f expressions of type T_1, then

   $$\mathbf{iterate}(e, v_T : T, v_{T_1} : T_1 = e_1 \mid f)$$

   is an OCL expression of type T_1. **iterate** is a built-in operation for all collection types. It is most natural for sequences but also applicable on sets and bags. Its concrete syntax is $e \to \mathbf{iterate}(v_T : T, v_{T_1} : T_1 = e_1 \mid f)$.
6. The usual **let** construct for introducing abbreviations may be freely used in OCL expressions. In the formal treatment we assume that all abbreviations have been resolved.
7. We do not exclude, that within an expression e occuring as a **pre** or **post** condition in the context of an operator *opname*, the function symbol f_{opname} attached to *opname* does occur.

4 Semantics of OCL

We will now continue the definition of $\mathcal{M}_D^{OCL} = (M_D^{OCL}, I^{OCL})$ begun in Subsection 3.1. Remember that we assume a fixed snapshot D of a UML class diagram $\mathcal{D}$. From $\mathcal{D}$ we read off the set of sort and function symbols, $S_{\mathcal{D}}$ and $F_{\mathcal{D}}$, while D may be described by the many-sorted structure $\mathcal{M}_D = (M_D, I)$. We will again take the liberty to omit the superscript OCL, when no confusion can arise. In Subsection 3.1 $I(e)$ was defined for all OCL type expressions. The OCL expression e may contain free variables. The meaning of e then depends on the values assigned to these free variables. This is the job of the function β: for every variable v in e of type S we require $\beta(v)$ to be an element in $I(S)$ and we will explain how to calculate recursively the value of $I_\beta(e)$. The initial cases (Clause 1 to 4) in Definition 10 are easy to deal with and likewise the induction steps for clauses 7 and 8 are obvious. We concentrate here on clause 5 and postpone clause 6 for later.

Definition 13 (Semantics of model properties).

1. *Let f_a in $F_\mathcal{D}$ be the function symbol attached to an attribute a of the class T (see Definition 4). Let further e be an OCL expression with $type(e) \ll_\mathcal{D}^{OCL} T$. Then $I_\beta(f(e)) = I(f_a)(I_\beta(e))$.*
2. *Let $f = f_{r,e_1,e_2}$ be a function symbol in $F_\mathcal{D}$ attached to the association r and the association ends e_1, e_2 with argument type T and e an OCL expressions with $type(e) \ll_\mathcal{D}^{OCL} T$. Then $I_\beta(f(e)) = I(f_{r,e_1,e_2})(I_\beta(e))$.*

Next we should give the semantics for all pre-defined operations of OCL. We restrict ourselves to a few typical cases.

Definition 14 (Semantics of model properties).

1. *Consider the expression $e \equiv e_1 \to collect(c \mid e_2)$ (in functional notation $collect(e_1, c, e_2)$).*
   (a) *$type(e_2)$ is not a collection type.*
   In case $type(e_1) = Set(S)$ or $Bag(S)$:
   $$I_\beta(e) = \text{ the bag of elements } I_{\beta_c^a}(e_2) \text{ for all } a \in I_\beta(e_1)$$
   In case $type(e_1) = Sequence(S)$:
   $$I_\beta(e) = \langle I_{\beta_{c_1}^a}(e_2), \ldots, I_{\beta_{c_k}^a}(e_2)\rangle \text{ with } I_\beta(e_1) = \langle c_1, \ldots, c_k\rangle$$
   (b) *$type(e_2) = Set(T)$ or $Bag(T)$*
   $$I_\beta(e) = \bigcup\{I_{\beta_c^a}(e_2) \mid a \in I_\beta(e_1)\}$$
   In any case the union $\bigcup$ will result in a bag, i.e. multiple occurences will not be eliminated. The information what kind of a collection type e_1 did possess is lost, even in the case $type(e_1) = Sequence(S)$.
   (c) *$type(e_2) = Sequence(T)$*
   If $type(e_1) = Set(S)$ or $Bag(S)$ the definition of the previous item applies where $I_{\beta_c^a}(e_2)$ is treated as the bag of elements occuring in the sequence.
   If $type(e_1) = Sequence(S)$ then $I_\beta(e)$ is the concatenation of the sequences $I_{\beta_{c_i}^a}(e_2)$, with $1 \le i \le k$ and $I_\beta(e_1) = \langle c_1, \ldots, c_k\rangle$.

In these definitions we have used $\beta_c^a(x) = \begin{cases} \beta(x) & \text{if } x \neq c \\ a & \text{if } x = c \end{cases}$

Definition 15. *Let v be the only free variable that may occur in e, and β_m the variable assignment with $\beta_m(v) = m$.*

The constraint **context** *t* **inv :** *e*
is true in $\mathcal{M}_D^{OCL} = (M_D, I)$ if for every $m \in I(T)$: $I_{\beta_m}(e) = \mathbf{true}$.

The meaning of constraints on methods is defined with respect to a pair of snapshots D_1, D_2 for the same diagram $\mathcal{D}$. These lead to a pair of structures of the form: $(\mathcal{M}^{OCL}_{D_1}, \mathcal{M}^{OCL}_{D_2}) = ((M_{D_1}, I_1),(M_{D_2}, I_2))$. We will further assume that $M_{D_1} \subseteq M_{D_2}$. This amounts to the stipulation that instances once created cannot be removed.

Definition 16. *The constraint*

> **context** *(c :) typeName* :: *opname*(p_1: *type*$_1$; ... ;p_k: *type*$_k$):*rtype*
>   **pre** : e_1
>   **post** : e_2

is true in $(\mathcal{M}^{OCL}_{D_1}, \mathcal{M}^{OCL}_{D_2})$ *if for every* β

$$I_{1,\beta}(e_1) = \mathbf{true} \textit{ implies } I_{2,\beta}(e_2) = \mathbf{true}$$

Here β *ranges over all functions from the set* $\{c, p_1, \ldots, p_k\}$ *of variables into the universe* M_{D_1} *satisfying the typing restrictions.*

It is in Definition 14 clause 2 and 3 that flattening takes place and the creation of sets of sets or sequences of sequences is avoided.

The standard [8, Subsection 7.5.5] allows to access association ends of multiplicity [0..1] both as sets and as elements. There is no clear way in our setting to mimick this overloading. We even doubt its practicality.

5 Future Research

We are presently working on a translation of OCL into Dynamic Logic, which is the input language of the theorem prover in the KeYproject. Special care has to be taken in the translation of the @*pre* suffix. It is quite easy to use partial functions in a semantics description, as we have done here. A deduction calculus for reasoning about partial functions in the OCL framework still has to be decided on.

5.1 Acknowledgements

Many thanks are due to Wolfgang Ahrendt, Bernhard Beckert, Joao Marcos, and Reiner Hähnle, for their generous help in preparing this paper.

References

1. T. Baar and R. Hähnle. An integrated metamodel for OCL types. In R. France, B. Rumpe, and J. Whittle, editors, *Proc. OOPSLA 2000 Workshop Refactoring the UML: In Search of the Core, Minneapolis/MI, USA*, Oct. 2000.
2. M. Fowler and K. Scott. *UML Destilled. Applying the Standard Object Modeling Language.* Addison-Wesley, 1997.
3. M. Gogolla and M. Richters. On constraints and queries in UML. In Schader and Korthaus [14], pages 109–121.
4. A. Hamie. A formal semantics for checking and analysing UML models. In L. Andrade, A. Moreira, A. Deshpande, and S. Kent, editors, *Proceedings of the OOPSLA'98 Workshop on Formalizing UML. Why? How?*, 1998.
5. A. Hamie, J. Howse, and S. Kent. Navigation expressions in object-oriented modelling. In *FASE'98*.

6. A. Hamie, J. Howse, and S. Kent. Interpreting the object constraint language. In *Proceedings of Asia Pacific Conference in Software Engineering*. IEEE Press, July 1998.
7. L. Mandel and M. V. Cengarle. On the expressive power of OCL. In *FM'99 - Formal Methods. World Congress on Formal Methods in the Development of Computing Systems, Toulouse, France, September 1999. Proceedings, Volume I*, volume 1708 of *LNCS*, pages 854–874. Springer, 1999.
8. OMG. Object constraint language specification, version 1.3. chapter 7 in [9]. OMG Document ad970808, September 1999.
9. OMG. OMG unified modeling language spezification, version 1.3. OMG Document, June 1999.
10. OMG. OMG unified modeling language spezification, version 1.3. OMG Document, March 2000.
11. M. Richters and M. Gogolla. On formalizing the UML object constraint language OCL. In T. W. Ling, S. Ram, and M. L. Lee, editors, *Proc. 17th Int. Conf. Conceptual Modeling (ER'98)*, pages 449–464. Springer, Berlin, LNCS 1507, 1998.
12. M. Richters and M. Gogolla. A metamodel for OCL. volume 1723 of *LNCS*, pages 156–171. Springer, 1999.
13. J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1998.
14. M. Schader and A. Korthaus, editors. *The Unified Modeling Language: technical aspects and applications*. Physica-Verlag, 1998.
15. M. Vaziri and D. Jackson. Some shortcomings of ocl, the object constraint language of uml. response to object management group's request for information on uml 2.0. Technical report, MIT Laboratory for Computer science, December 1999.
16. J. Warmer and A. Kleppe. *The Object Constraint Language: Precise Modelling with UML*. Object Technology Series. Addison-Wesley, Reading/MA, 1999.