# A Proposal for a Formal OCL Semantics in Isabelle/HOL

**Extended Version**

Achim D. Brucker and Burkhart Wolff

Institut für Informatik, Albert-Ludwigs-Universität Freiburg
Georges-Köhler-Allee 52, D-79110 Freiburg, Germany
{brucker,wolff}@informatik.uni-freiburg.de
http://www.informatik.uni-freiburg.de/~{brucker,wolff}

**Abstract** We present a formal semantics as a conservative shallow embedding of the Object Constraint Language (OCL). OCL is currently under development within an open standardization process within the OMG; our work is an attempt to accompany this process by a proposal solving open questions in a consistent way and exploring alternatives of the language design. Moreover, our encoding gives the foundation for tool supported reasoning over OCL specifications, for example as basis for test case generation.
**Keywords:** Isabelle, OCL, UML, shallow embedding, testing

## 1 Introduction

The Unified Modeling Language (UML) [1] has been widely accepted throughout the software industry and is successfully applied to diverse domains [2]. UML is supported by major CASE tools and integrated into a software development process model that stood the test of time. The Object Constraint Language (OCL) [3, 4, 5] is a textual extension of the UML. OCL is in the tradition of data-oriented formal specification languages like Z [6] or VDM [7]. For short, OCL is a three-valued Kleene-Logic with equality that allows for specifying constraints on graphs of object instances whose structure is described by UML class diagrams.

In order to achieve a maximum of acceptance in industry, OCL is currently developed within an open standardization process by the OMG. Although the OCL is part of the UML standard since version 1.3, at present, the official OCL standard 1.4 concentrates on the concrete syntax, covers only in parts the well-formedness of OCL and handles nearly no formal semantics. So far, the description of the OCL is merely an informal requirement analysis document with many examples, which are sometimes even contradictory.

Consequently, there is a need[1] for both software engineers and CASE tool developers to clarify the concepts of OCL formally and to put them into perspective of more standard semantic terminology. In order to meet this need, we

[1] This work was partially funded by the OMG member Interactive Objects Software GmbH (www.io-software.com).

C. Muñoz, S. Tahar, V. Carreño (Eds.): TPHOLs 2002, LNCS 2410, pp. 99–114, 2003.

started to provide a conservative embedding of OCL into Isabelle/HOL. As far as this was possible, we tried to follow the design decisions of OCL 1.4 in order to provide insight into the possible design choices to be made in the current standardization process of version 2.0.

Attempting to be a "practical formalism" [4], OCL addresses software developers who do not have a strong mathematical background. Thus, OCL deliberately avoids mathematical notation; rather, it uses a quite verbose, programming language oriented syntax and attempts to hide concepts such as logical quantifiers. This extends also to a design rationale behind the semantics: OCL is still viewed as an object-oriented assertion language and has thus more similarities with an object-oriented programming language than a conventional specification language. For example, standard library operators such as "concat" on sequences are defined as *strict operations* (i.e. they yield an explicit value *undefined* as result whenever an argument is undefined), only bounded quantifiers are admitted, and there is a tendency to define infinite sets away wherever they occur. As a result, OCL has a particularly executable flavor which comes handy when generating code for assertions or when animating specifications.

Object-oriented languages represent a particular challenge for the "art of embedding languages in theorem provers" [8]. This holds even more for a shallow embedding, which we chose since we aim at reasoning in OCL specifications and not at meta-theoretic properties of our OCL representation. In a shallow embedding, the types of OCL language constructs have to be represented by types of HOL and concepts such as undefinedness, mutual recursion between object instances, dynamic types, and extensible class hierarchies have to be handled.

In this paper, we present a new formal model of OCL in form of a conservative embedding into Isabelle/HOL that can cope with the challenges discussed above. Its modular organization has been used to investigate the interdependence of certain language features (method recursion, executability, strictness, smashing, flattening etc.) in order to provide insight into the possible design choices for the current design process [9]. We extend known techniques for the shallow representation of object orientation and automated proof techniques to lift lemmas from the HOL-library to the OCL level. As a result, we provide a first calculus to formally reason over OCL specifications and provide some foundation for automated reasoning in OCL.

This paper proceeds as follows: After a introduction into our running example using UML/OCL, we will guide through the layers of our OCL semantics, namely the object model and resulting semantic universes, the states and state relations, the OCL logic and the OCL library. It follows a description of automated deduction techniques based on derived rules, let it be on modified tableaux-deduction techniques or congruence rewriting. We will apply these techniques in a paradigmatic example for test-case generation in a black-box test setting.

## 2 A Guided Tour Through UML/OCL

The UML provides a variety of diagram types for describing dynamic (e.g. state charts, activity diagrams, etc.) and static (class diagrams, object diagrams, etc.) system properties. One of the more prominent diagram types of the UML is the

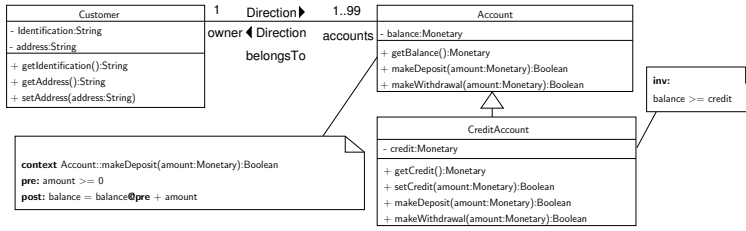


**Figure 1.** Modeling a simple banking scenario with UML

*class diagram* for modeling the underlying data model of a system in an object oriented manner. The class diagram in our running example in Fig. 1 illustrates a simple banking scenario describing the relations between the classes Customer, Account and its specialization CreditAccount. To be more precise, the relation between data of the classes Account and CreditAccount is called *subtyping*. A class does not only describe a set of record-like data consisting of *attributes* such as balance but also functions (*methods*) defined over the classes data model.

It is characteristic for the object oriented paradigm, that the functional behavior of a class and all its methods are also accessible for all subtypes; this is called *inheritance*. A class is allowed to redefine an inherited method, as long as the method interface does not change; this is called *overwriting*, as it is done in the example for the method makeWithdrawal().

It is possible to model relations between classes (*association*), possibly constrained by *multiplicities*. In Fig. 1, the association belongsTo requires, that every instance of Account is associated with exactly one instance of Customer. It characterizes that every account has a unique owner. In the other direction, the association models that an instance of class Customer is related to a set of instances of class Account or its subtypes, the size of this set is limited to number between one and 99. Associations were represented by introducing implicit set-valued attributes into the objects, while multiplicity were mapped to suitable data invariants. In the following, we assume that associations have already been "parsed away".

Understanding OCL as a data-oriented specification formalism, it seems natural to refine class diagrams using OCL for specifying invariants, pre- and post-conditions of methods. For example, see Fig. 1, where the specification of the method makeWithdrawal() is given by its pair of pre- and postcondition.

In UML class members can contain attributes of the type of the defining class. Thus, UML can represent (mutually) recursive data types. Moreover, OCL

introduces also recursively specified methods [3]; however, at present, a dynamic semantics of a method call is missing (see [9] for a short discussion of the resulting problems).

## 3 Representing OCL in Isabelle/HOL

OCL formulae are built over expressions that access the underlying state. In postconditions, path-expressions can access the *current* and the *previous state*, e.g. balance = balance**@pre** + amount. Accesses on both states may be arbitrarily mixed, e.g. **self**.x**@pre**.y denotes an object, that was constructed by dereferencing in the previous state and selecting attribute x in it, while the next dereferencing step via y is done in the current state. Thus, method specifications represent state transition relations built from the conjunction of pre and post condition, where the state consists of a graph of object instances whose type is defined by the underlying class diagram. Since the fundamental CL-operator allInstances allows for the selection of all instances (objects) of a certain class, there must be a means to reconstruct their dynamic types in the object graph.

In a shallow embedding, the key question arises how to represent the static type structure of the objects *uniformly* within a state or as argument of a dynamic type test is_T. Constructions like "the universe of all class diagram interpretations" are too large to be represented in the simple type system underlying higher-order logic.

Our solution is based on the observation that we need not represent *all* class diagram interpretations inside the logic; it suffices if we can provide an extralogical mechanism that represents any concrete class hierarchy and that allows for extensions of any given class hierarchy. For practical reasons, we require that such an *extension* mechanism is logically conservative both in the sense that only definitional axioms are used and that all existing proofs on data of the former class hierarchy remain valid for an extended one.

Based on these considerations, *HOL-OCL* is organized in several layers:

- a *semantic coding scheme* for the *object model* layer along the class-hierarchy,
- the *system state* and *relations* over it, forming the denotational domain for the semantics of methods,
- the *OCL logic* for specifying state relations or class invariants,
- the *OCL library* describing predefined basic types.

### 3.1 The Encoding of Extensible Object Models

The main goals of our encoding scheme is to provide typed constructors and accessor functions for a given set of classes or enumeration types to be inserted in a previous class hierarchy. The coding scheme will be represented in two steps: in this section, we will describe *raw* constructors and accessors, while in Sec. 3.2 a refined scheme for accessors is presented.

The basic configuration of any class hierarchy is given by the OCL standard; the library for this basic configuration is described in Sec. 3.5.

**Handling Undefinedness.** In the OCL standard 1.4, the notion of explicit undefinedness is part of the language, both for the logic and the basic values.

> Object Constraint Language Specification [3] (version 1.4), page 6–58
>
> Whenever an OCL-expression is being evaluated, there is the possibility that one or more queries in the expression are undefined. If this is the case, then the complete expression will be undefined.

This requirement postulates the strictness of all operations (the logical operators are explicit exceptions) and rules out a modeling of undefinedness via underspecification. Thus, the language has a similar flavor than LCF or SPECTRUM [10] and represents a particular challenge for automated reasoning.

In order to handle undefinedness, we introduce for each type $\tau$ a *lifted type* $\tau_\perp$, i.e. we introduce a special type constructor. It adds to each given type an additional value $\perp$. The function $\lfloor\_\rfloor : \alpha \Rightarrow \alpha_\perp$ denotes the injection, the function $\lceil\_\rceil : \alpha_\perp \Rightarrow \alpha$ its inverse. Moreover, we have the case distinction function case_up $c\ f\ x$ that returns $c$ if $x = \perp$ and $f\ k$ if $x = \lfloor k \rfloor$. We will also write $\mathrm{case}\, x\ \mathrm{of} \lfloor k \rfloor \Rightarrow f\ k \mid \perp \Rightarrow c$.

Note that the definition of lifted types leads to the usual construction of *flat cpo's* well known from the theory of complete partial orders (*cpo*) and denotational semantics [11]. For the sake of simplification, we avoid a full-blown cpo-structure here (while maintaining our semantics "cpo-ready") and define only a tiny fragment of it that provides concepts such as *definedness* $\mathrm{DEF}(x) \equiv (x \neq \perp)$ or *strictness of a function* $\mathrm{is\_strict}\, f \equiv (f\perp = \perp)$.

**Managing Holes in Universes.** Since objects can be viewed as records consisting of *attributes*, and since the object alternatives can be viewed as variants, it is natural to construct the "type of all objects", i.e. the semantic universe $\mathscr{U}_x$ corresponding to a certain class hierarchy $x$, by Cartesian products and by type sums (based on the constructors $\mathrm{Inl} : \alpha \Rightarrow \alpha + \beta$ and $\mathrm{Inr} : \beta \Rightarrow \alpha + \beta$ from the Isabelle/HOL library). In order to enable extensibility, we provide systematically polymorphic variables — the "holes" — into a universe that were filled when extending a class hierarchy.

In our scheme, a class can be extended in two ways: either, an alternative to the class is added *at the same level* which corresponds to the creation of an alternative subtype of the supertype (the *β-instance*), or a class is added *below* which corresponds to the creation of a subtype (the *α-instance*). The insertion of a class corresponds to *filling a hole* $\nu$ *by a record* $T$ is implemented by the particular type instance:

$$\nu \mapsto ((T \times \alpha_\perp) + \beta)$$

As a consequence, the universe $\mathscr{U}^2$ in Fig. 2 is just a type instance of $\mathscr{U}^1$, in particular: an $\alpha$-extension with $C$ $\beta$-extended by $D$. Thus, properties proven over $\mathscr{U}^1$ also holds for $\mathscr{U}^2$.

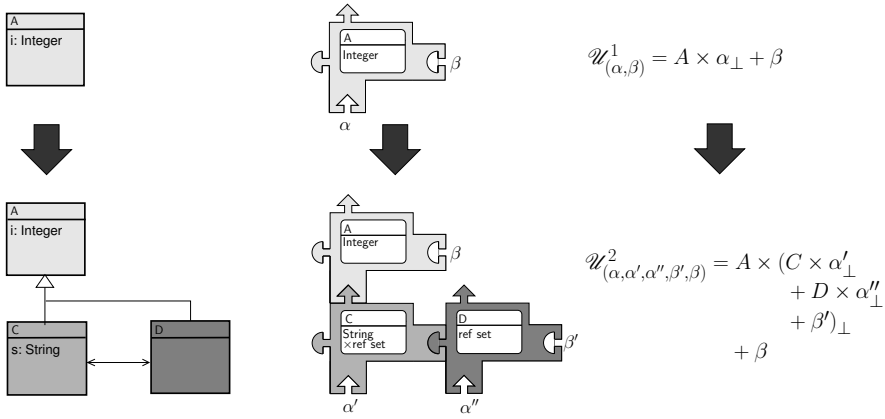


**Figure 2.** Extending Class Hierarchies and Universes with Holes

The *initial* universe corresponding to the minimal class hierarchy of the OCL library consists of the real numbers, strings and bool. It is defined as follows:

$$\text{Real} = \text{real}_\perp \quad \text{Boolean} = \mathit{bool}_\perp \quad \text{String} = \text{string}_\perp \quad \mathsf{OclAny}_\alpha = \alpha_\perp$$
$$\mathscr{U}_\alpha = \text{Real} + \text{Boolean} + \text{String} + \mathsf{OclAny}_\alpha$$

Note that the $\alpha$-extensions were all lifted, i.e. additional $\perp$ elements were added. Thus, there is a uniform way to denote "closed" objects, i.e. objects whose potential extension is not used. Consequently, it is possible to determine the dynamic type by testing for closing $\perp$'s. For example, the OclAny type has exactly one object represented by $\text{Inr}(\text{Inr}(\text{Inr}\,\perp))$ in any universe $\mathscr{U}_\alpha$.

Note, moreover, that all types of attributes occurring in the records $A$,$C$ or $D$ may contain basic types, and sets, sequences or bags over them, but not references to the types induced by class declarations. These references were replaced the abstract type ref to be defined later; thus, recursion is not represented at the level of the universe construction, that just provides "raw data".

**Outlining the Coding Scheme.** Now we provide raw constructors, raw accessors, and tests for the dynamic types over the data universe.

The idea is to provide for each class $T$ with attributes $t_1 : \tau_1, \ldots, t_n : \tau_n$ a type $T = \tau_1 \times \cdots \times \tau_n$ and a constructor $\text{mk\_T} : T \Rightarrow \mathscr{U}_x$, which embeds a record of type $T$ into the actual version of the universe (for example mk_Boolean with type $\text{Boolean} \Rightarrow \mathscr{U}_\alpha$ is defined by $\text{mk\_Boolean} \equiv \text{Inr} \circ \text{Inl}$). Accordingly, there is a test $\text{is\_T} : \mathscr{U}_x \Rightarrow \mathit{bool}$ that checks if an object in the universe is embedded as $T$, and an accessor $\text{get\_T} : \mathscr{U}_x \Rightarrow T$ that represents the corresponding projection. Finally, a constant $T : \mathscr{U}_x$ set is provided that contains the *characteric set* of $T$ in the sense of a set of all objects of class $T$.

Data invariants $I$ are represented by making the constructor partial w.r.t. $I$, i.e. the constructor will be defined only for input tuples $T$ that fulfill $I$; correspondingly, the test for the dynamic type is also based on $I$.

At present, we encode our examples by hand. The task of implementing a compiler that converts representations of UML-diagrams (for example, formats produced by ArgoUML) is desirable but not in the focus of this research.

### 3.2 System State

**Basic Definitions.** The task of defining the state or state transitions is now straight-forward: We define an abstract type ref for "references" or "locations", and a state that is a partial mapping from references to objects in a universe:

$$\begin{aligned} \textbf{types} \quad & \text{ref} \\ & \sigma\,\text{state} = \text{ref} \Rightarrow \alpha\,\text{option} \\ & \sigma\,\text{st} \quad = \sigma\,\text{state} \times \sigma\,\text{state} \end{aligned}$$

Based on state, we define the only form of universal quantification of OCL: the operator allInstances extracts all objects of a "type" — represented by its characteristic set — from the current state. The standard specifies allInstances as being undefined for *Integer* or Real or String. Thus, infinite sets are avoided in an ad-hoc manner. However, nothing prevents from having infinite states; and we did not enforce finiteness by additional postulates.

$$\begin{aligned} & \mathsf{allInstances} : [\mathscr{U}_\alpha\,\text{set}, \mathscr{U}_\alpha\,\text{st}] \Rightarrow \beta\,\text{Set} \\ & \mathsf{allInstances}\,type \equiv \lambda(s, s') \bullet \textbf{if}(type = \mathit{Integer} \vee\ type = \text{Real} \vee type = \text{String}) \\ & \qquad \textbf{then} \perp \textbf{else if}(type = \text{Boolean}) \\ & \qquad\qquad \textbf{then}\lfloor\text{Boolean}\rfloor\ \textbf{else}\lfloor type \cap (\text{ran}\, s')\rfloor \end{aligned}$$

Defining OCL operators like oclIsNew : $\mathscr{U}_\alpha \Rightarrow \mathscr{U}_\alpha\,\text{st} \Rightarrow$ Boolean or oclIsTypeOf is now routine; the former checks if an object is defined in the current but not in the previous state, while the latter redefines the test is_T of Sec. 3.1.

**The HOL Type of OCL Expressions.** Functions from state transition pairs to lifted OCL values will be the denotational domain of our OCL semantics. From a transition pair, all values will be extracted (via path expressions), that can be passed as arguments to to basic operations or user-defined methods. More precisely, all expressions with OCL type $\tau$ will be represented by an *HOL-OCL* expression of type $V_\alpha(\tau_\perp)$ defined by:

$$\textbf{types} \quad V_\alpha(\theta) = \mathscr{U}_\alpha\,\text{st} \Rightarrow \theta$$

where $\alpha$ will represent the type of the state transition. For example, all *logical HOL-OCL* expressions have the type $V_\gamma$(Boolean) (recall that Boolean is the lifted type *bool* from HOL).

As a consequence, all operations and methods embedded into *HOL-OCL* will have to pass the context state transition pair to its argument expressions, collect the values according their type, and compute a result value. For example, let a function $f$ have the OCL type $\tau_1 \times \cdots \times \tau_n \Rightarrow \tau_{n+1}$, then our representation $f'$ will have the type $V_\alpha(\tau_1) \times \cdots \times V_\alpha(\tau_n) \Rightarrow V_\alpha(\tau_{n+1})$. Now, when defining $f'$, we proceed by $f'(e_1, \ldots, e_n)(c) = E(e_1 c, \ldots, e_n c)$ for some $E$ and some context transition $c$. We call the structure of definitions *state passing*. Functions with one argument of this form are characterized semantically:

$$\text{pass}\, f = \exists E \bullet \forall X\ c \bullet f\ X\ c = E(X\ c)$$

Being state passing will turn out to be an important invariant of our shallow semantics and will be essential for our OCL calculus. The conversion of a function $f$ into $f'$ in state passing style will be called the *lifting* of $f$ (which should not be confused with the lifting on types of Sec. 3.1).

**A Coding Scheme for State-Based Accessors.** Now we define the accessor functions with the "real OCL signature", that can be used to build up path expressions, on the basis of raw accessors. Two problems have to be solved: first, references to class names occurring in types of attributes must be handled (i.e. ref set in raw accessors types must be mapped to Set($A$)), and second, raw accessors must be lifted to state passing style. In a simple example, an accessor $x$ of type $\tau$ in a class $C$ must have the HOL type $x : \mathscr{U}_x \Rightarrow V_x(\tau)$ which is normally achieved by wrapping the raw accessor $x_0$ in an additional abstraction: $x\ u = \lambda c \bullet x_0\ u$. If the class has a class invariant, a test for the invariant must be added (violations are considered as undefinedness and therefore treated like to undefined references into the state). If the accessor yields an OCL-type with references to other classes (e.g. in Set($A$)), these references must be accessed and inserted into the surrounding collection; this may involve smashing (see the discussion of collection types in Sec. 3.5).

Following this extended code scheme, we can define conservatively new accessors over some extended universe whenever we extend a class hierarchy; this allows for modeling mutual data recursion that is introduced by extension while maintaining static typechecking.

### 3.3 Encoding our Example

In order to encode our UML model in Fig. 1, we declare the type for the class Account (we skip CreditAccount and Customer). An account is a tuple describing the balance (of type Monetary) and the encoded association end '*owner*' (of type ref set). For our first universe, with the two "holes" $\alpha$ (for extending "below") and $\beta$ (for extending on "the same level"), we define:

$$\begin{aligned} \textbf{types} \quad \text{Account\_type} &= \text{Monetary} \times \text{ref set} \\ \text{Account} &= \text{Account\_type}_\perp \\ \mathscr{U}^1_{(\alpha,\beta)} &= \mathscr{U}_{(\text{Account\_type} \times \alpha_\perp + \beta)} \end{aligned}$$

We need the raw constructor for an account object. Note that this function "lives" in the universe $\mathscr{U}^3_{(\alpha',\alpha'',\beta',\beta)}$ which contains *all* classes from Fig. 1.

$$\begin{aligned}&\text{mk\_Account} \;:\; \text{Account\_type} \Rightarrow \mathscr{U}^3_{(\alpha',\alpha'',\beta',\beta)}\\ &\text{mk\_Account} \equiv \text{mk\_OclAny} \circ \text{lift} \circ \text{Inl} \circ \lambda\, x \bullet (x, \bot)\end{aligned}$$

In the next step, we need to define the the accessors for the attribute. As an example, we present the definition for accessing the association end *owner* (of type Customer) in the current state $s'$:

$$\begin{aligned}&.\text{owner} \;:\; \mathscr{U}^3_{(\alpha',\alpha'',\beta',\beta)} \Rightarrow V_{(\alpha',\alpha'',\beta',\beta)}(\text{Customer})\\ &(obj.\text{owner}) \equiv (\lambda(s,s') \bullet \text{up\_case}(\text{lift} \circ ((\text{op }“)(\lambda\, x \bullet \text{option\_Case}\, \bot\\ &\qquad \text{get\_Customer}(s'\; x))) \circ \text{snd})(\bot)(\text{get\_Account}\, obj))\end{aligned}$$

Analogously, we define the association end *owner@pre* for accessing the value in the previous state $s$ (which is the only difference to the definition above):

$$\begin{aligned}&.\text{owner@pre} \;:\; \mathscr{U}^3_{(\alpha',\alpha'',\beta',\beta)} \Rightarrow V_{(\alpha',\alpha'',\beta',\beta)}(\text{Customer})\\ &(obj.\text{owner@pre}) \equiv (\lambda(s,s') \bullet \text{up\_case}(\text{lift} \circ ((\text{op }“)(\lambda\, x \bullet \text{option\_Case}\, \bot\\ &\qquad \text{get\_Customer}(s\; x))) \circ \text{snd})(\bot)(\text{get\_Account}\, obj))\end{aligned}$$

Note, that accessor functions are lifted, e.g. they operate over a previous and current state pair $(s, s')$.

### 3.4 OCL Logic

We turn now to a key chapter of the OCL-semantics: the logics. According to the OCL standard (which follows SPECTRUM here), the logic operators have to be defined as Kleene-Logic, requiring that any logical operator reduces to a defined logical value whenever possible.

In itself, the logic will turn out to be completely independent from an underlying state transition pair or universe and is therefore valid in all universes. An OCL formula is a function that is either true, false or undefined depending on its underlying state transition. Logical expressions are just special cases of OCL expressions and must produce Boolean values. Consequently, the general type of logical formulae must be:

$$\textbf{types} \quad \text{BOOL}_\alpha = V_\alpha(\text{Boolean})$$

The logical constants $\mathsf{true}$ resp. $\mathsf{false}$ can be defined as constant functions, that yield the lifted value for meta-logical undefinedness, truth or falsehood, i.e. the HOL values of the HOL type bool. Moreover, the predicate is_def decides for any OCL expression $X$ that its value (evaluated in the context $c$) is defined:

$$\begin{aligned}\textbf{constdefs} \quad &\text{is\_def} : V_\alpha(\beta_\perp) \Rightarrow \text{BOOL}_\alpha \quad \text{is\_def}\, X \equiv \lambda\, c \bullet \lfloor \text{DEF}(X\; c) \rfloor\\ &\bot_{\mathscr{L}} : \text{BOOL}_\alpha \qquad\qquad \bot_{\mathscr{L}} \equiv \lambda\, x \bullet \lfloor \bot \rfloor\\ &\mathsf{true} : \text{BOOL}_\alpha \qquad\qquad \mathsf{true} \equiv \lambda\, x \bullet \lfloor \text{True} \rfloor\\ &\mathsf{false} : \text{BOOL}_\alpha \qquad\qquad \mathsf{false} \equiv \lambda\, x \bullet \lfloor \text{False} \rfloor\end{aligned}$$

The definition of the strict not and and, or, and implies are straight-forward:

$$\begin{aligned}
&\mathsf{not} : \mathrm{BOOL}_\alpha \Rightarrow \mathrm{BOOL}_\alpha \quad \mathsf{not}\, S \equiv \lambda\, c \bullet \mathbf{if}\, \mathrm{DEF}(S\ c)\, \mathbf{then} \lfloor \neg \lceil S\ c \rceil \rfloor\, \mathbf{else}\, \bot \\
&\mathsf{and} : [\mathrm{BOOL}_\alpha, \mathrm{BOOL}_\alpha] \Rightarrow \mathrm{BOOL}_\alpha \\
&S\, \mathsf{and}\, T \equiv \lambda\, c \bullet \mathbf{if}\, \mathrm{DEF}(S\ c)\, \mathbf{then}\, \mathbf{if}\, \mathrm{DEF}(T\ c)\, \mathbf{then} \lfloor \lceil S\ c \rceil \wedge \lceil T\ c \rceil \rfloor\, \mathbf{else} \\
&\qquad\qquad \mathbf{if}(S\ c = \lfloor \mathrm{False} \rfloor)\, \mathbf{then} \lfloor \mathrm{False} \rfloor\, \mathbf{else}\, \bot\, \mathbf{else} \\
&\qquad\qquad\qquad \mathbf{if}(T\ c = \lfloor \mathrm{False} \rfloor)\, \mathbf{then} \lfloor \mathrm{False} \rfloor\, \mathbf{else}\, \bot
\end{aligned}$$

From these definitions, the following rules of the truth table were derived:

| $a$ | not $a$ |
|---|---|
| true | false |
| false | true |
| $\bot_{\mathscr{L}}$ | $\bot_{\mathscr{L}}$ |

| $a$ | $b$ | $a$ and $b$ |
|---|---|---|
| false | false | false |
| false | true | false |
| false | $\bot_{\mathscr{L}}$ | false |

| $a$ | $b$ | $a$ and $b$ |
|---|---|---|
| true | false | false |
| true | true | true |
| true | $\bot_{\mathscr{L}}$ | $\bot_{\mathscr{L}}$ |

| $a$ | $b$ | $a$ and $b$ |
|---|---|---|
| $\bot_{\mathscr{L}}$ | false | false |
| $\bot_{\mathscr{L}}$ | true | $\bot_{\mathscr{L}}$ |
| $\bot_{\mathscr{L}}$ | $\bot_{\mathscr{L}}$ | $\bot_{\mathscr{L}}$ |

Based on these basic equalities, it is not difficult to derive with Isabelle the laws of the perhaps surprisingly rich algebraic structure of Kleene-Logics: Both and and or enjoy not only the usual associativity, commutativity and idempotency laws, but also both distributivity and de Morgan laws. It is essentially this richness and algebraic simplicity that we will exploit in the example in Sec. 5.

OCL needs own equalities, a logical one called *strong equality* ($\hat{=}$) and a strictified version of it called *weak equality* ($\doteq$) that is executable. They have the type $[V_\alpha(\beta), V_\alpha(\beta)] \Rightarrow \mathrm{BOOL}_\alpha$ and are defined similarly to and above based on the standard HOL equality.

### 3.5 The Library: OCL Basic Data Types

The library provides operations for Integer, Real (not supported at present) and Strings. Moreover, the parametric data types Set, Sequence and Bag with their functions were also provided; these were types were grouped into a class "Collection" in the standard. At present, the standard prescribes only ad-hoc polymorphism for the operations of the library and not *late binding*.

Since the standard suggests a uniform semantic structure of all functions in the library, we decided to make the uniformity explicit and to exploit it in the proof support deriving rules over them.

In the library, all operations are *lifted*, *strictified* and (as far as collection functions are concerned; see below) *smashed* versions of functions from the HOL library. However, *methodification* (i.e. introduction of late binding), is not needed here due to the standards preference of ad-hoc polymorphism. This will be described as semantic option for user-defined methods in Sec. 3.6.

The generic functions for $\mathrm{lift}_2$ and strictify are defined as follows:

$$\begin{aligned}
\mathrm{strictify}\ f\ x &\equiv \mathrm{case}\, x\, \mathrm{of} \lfloor v \rfloor \Rightarrow (f\ v) | \bot \Rightarrow \bot \\
\mathrm{lift}_2\ f &\equiv (\lambda\, X\ Y\ st \bullet f(X\ st)(Y\ st))
\end{aligned}$$

According to this definition, $\mathrm{lift}_2$ converts a function of type $[\alpha, \beta] \Rightarrow \gamma$ to a function of type $[V_\sigma(\alpha), V_\sigma(\beta)] \Rightarrow V_\sigma(\gamma)$.

**A Standard Class: Integer.** Based on combinators like strictify and $\text{lift}_2$, the definitions of the bulk of operators follow the same pattern exemplified by:

$$\begin{array}{ll} \textbf{types} & \text{INTEGER}_\alpha = V_\alpha(\mathit{Integer}) \\ \textbf{defs} & \quad \text{op} + \equiv \text{lift}_2(\text{strictify}(\lambda\, x : \text{int} \bullet \text{strictify}(\lambda\, y \bullet \lfloor x + y \rfloor))) \end{array}$$

**A Collection Class: Set.** For collections, the requirement of having strict functions must consequently be extended to the constructors of sets. Since it is desirable to have in data types only denotations that were "generated" by the constructors, this leads to the concept of *smashing* of all collections. For example, smashed sets are identified with $\bot_{\mathscr{L}}$ provided one of their elements is $\bot$: $\{a, \bot_{\mathscr{L}}\} = \bot_{\mathscr{L}}$; Analogously, smashed versions of $Bags$, $Seq$ or $Pairs$ can be defined. Smashed sets directly represent the execution behavior in usual programming languages such as Java. We omit the details of the construction of smashed sets here for space reasons; apart from smashing arguments, the definitions of set operations such as includes, excludes, union or intersection follow the usual pattern.

The OCL standard prescribes also a particular concept called *flattening*. This means for example that a set $\{a, \{b\}, c\}$ is identified as $\{a, b, c\}$. We consider flattening as a syntactic issue and require that a front-end "parses away" such situations and generates conversions.

### 3.6 Overloading and Recursive Method Invocation

So far, the standard 1.4 has avoided to prescribe any concrete operational behavior of a method invocation. In [4], the only suggestion made is the pragmatic guideline to follow Liskov's principle ([**?**]; i.e. an overloading method is essentially a refinement of the overloaded method). While we acknowledge that prescribing a concrete operational semantics of method invocation might result in incompatibilities to existing object oriented languages, we believe that underspecification ("the programming language may choose an arbitrary overloaded method that matches") is not a satisfactory solution for an object oriented specification language. This holds in particular for recursive method invocation.

In our approach, we will formalize the most common understanding of method invocation based on *late-binding* (as implemented in Java, C++): Take the *least* method of an object, i.e. the function with the least input domain that fits to the object the method is invoked at. Our treatment of methods is based on the following compilation scheme:

1. *declarations* of a method $m : A_1 \times A_n \to B$ in a class $A_0$ were mapped to a declaration of an *overloading table* $m_{tab} : A_0 set \to A_0 \to A_1 \times A_n \to B$,
2. *(overloading) implementations* were represented by a conservative axiom of the form $m_{tab} T \equiv E$ where the construction assures that $T$ is "fresh",
3. *invocations* of a method were represented by using a "methodify"- combinator that stamps late-binding semantics on a function call.

The overloading table resulting from a *declaration* of a method takes as first argument the (sub) type — represented by a set of values —, then the first argument (the "object" the method is applied on and which is required to belong to the type), and then the other arguments of the method.

*Overloading* is represented by a variant of a constant definition [2]: $m_{tab}$ is considered as a family of constants indexed by types characterized by their sets. Consequently, following the overall scheme of constant definitions (see [**?**] for more details), $m_{tab}T'$ may not occur in $E$; however $m_{tab}T$ on previously defined $T$ is admissible in $E$. Note that $T'$ may have a more special type that all previously declared $T$, i.e. it may be of $\mathscr{U}_{x'}$ which is an instance of $\mathscr{U}_x$. As a consequence, our mechanism can not explain full recursion *and* late-binding so far. For a general treatment, the abstraction of the method table from the method body is necessary: this extends the usual technique of semantics construction of recursive functions to methods based on late-binding (see below).

In OCL, which is centered around *method specifications*, method definitions to be used in overloadings must be created by choosing one solution of a method specification via Hilbert-operator.

As a prerequisite for *invocations*, we define a combinator $match :: [\alpha \text{ set} \Rightarrow \beta, \alpha] \Rightarrow \beta$ that handles the lookup in an overloading table $m_{tab}$ according to the object *obj* as follows:

$$\text{match } m_{tab}\ obj \equiv (\text{the}(m_{tab}(\text{LEAST } X : \alpha\,\text{set} \bullet X \in \text{dom}\, m_{tab} \wedge obj \in X)))$$

On the basis of the match-combinator, we define an operator "methodify" that turns a lifted function in the sense of the previous sections into a method that can be overridden by future class extensions. For the case of OCL, methodification has to be combined with lifting as usual.

Both the OCL standard 1.4 and the standard draft 2.0 allow *recursive methods* "as long as the recursion is not infinite". The documents make no clear statement what the nature of non-terminating recursive definition might be. Essentially two possibilities come to mind:

- It could be illegal OCL. However, since non-termination is undecidable, this would imply a notion of well-formedness, that is not machine-checkable. A variant of this approach is the restriction to well-founded recursion; however, this would require new syntactic and semantic concepts for OCL (well-founded orderings, **measure**-statements, etc.).
- It could be "undefined", i.e. $\perp_{\mathscr{L}}$. This is consistent with the *least fixpoint* in the theory of complete partial orderings (cpo; c.f. [11]). This idea has already been proposed by [12].

A recursive method like the famous factorial function in a class $C$:

```
context fac(n: Integer): Integer
post: if n=0 then result=1 else result=n*fac(n-1) endif
```

[2] Not supported by Isabelle; this definitions are encoded by axioms.

corresponds to an axiom that defines the method table for fac at the point $C$ as follows:

**axiom**

$$\text{M\_fac } C = \text{Some}(\text{fix}(\lambda \text{ fac } n \bullet \mathsf{if}\ n \doteq 0_\perp\ \mathsf{then}\ 1_\perp\ \mathsf{else}\ n \cdot \text{Fac (fac)}(n - 1_\perp)\ \mathsf{endif}))$$
$$\textbf{where } \text{Fac (fac)}(a) \equiv \lambda\, st \bullet \text{match (M\_fac}[C \mapsto \text{fac}])(a\ st)(a\ st)$$

where fix : $(\alpha :: cpo \Rightarrow \alpha) \Rightarrow \alpha$ is the least fixpoint operator of a denotational semantics theory such as Isabelle/HOLCF, for example, and M_fac is the overloading table of fac. According to representation of method invocation, we have the usual late-binding behavior: Let $f$ and $g$ be two mutually recursive methods in a class $A$, and $B$ be a subclass of $A$ in which $g$ is overridden with an method body still calling $A.f$. Now, if $B.g$ is invoked, there is a mutual recursion between $A.f$ and $B.g$ without any change of the code of $A.f$.

Interpreting recursive methods as least fixpoint requires to base all semantic domains of OCL on a cpo structure. Since any function has a cpo-structure provided its range has a cpo-structure, and since any lifted types have cpo-structure, this is possible for *HOL-OCL* and fairly easy. In order to execute or prove properties over recursive methods, it is additionally necessary that the method bodies are *continuous*; for *HOL-OCL* with all its strict (and therefore continuous) operations, this essentially boils down to the requirement that no strong equality is used in them, except just after *result*.

## 4 Towards Automated Theorem Proving in *HOL-OCL*

Based on derived rules, we will provide several calculi and proof techniques for OCL that are oriented towards Isabelle's powerful proof-procedures like *fast_tac* and *simp_tac*. While the former is geared towards natural deduction calculi, the latter is based on rewriting and built for reasoning in equational theories.

### 4.1 A natural deduction-Calculus for OCL

As a foundation, we introduce two notions of validity: a formula may be *valid for all transitions* or just *valid* (written $\vDash P$) or be *valid for a transition* $t$ (written $t \vDash P$). We can define these notions by $\vDash P \equiv P = \mathsf{true}$ or $t \vDash P \equiv P\ t = \mathsf{true}\ t$ respectively. Recall that a formula may neither be valid nor invalid in a state, it can be undefined:

$$\frac{\begin{array}{ccc}[st \vDash not(\text{is\_def}(A))] & [st \vDash not(A)] & [st \vDash A] \\ \vdots & \vdots & \vdots \\ R & R & R\end{array}}{R}$$

This rule replaces in a Kleene-Logic the usual *classical* rule. Note that $R$ may be an arbitrary judgment.

The core of the calculus consists of the more conventional rules like:

$$\frac{\models A \quad \models B}{\models A \,\mathsf{and}\, B} \qquad \frac{\models A \,\mathsf{and}\, B}{\models A} \qquad \frac{\models A \,\mathsf{and}\, B}{\models A} \qquad \frac{\models A \,\mathsf{and}\, B \qquad \begin{array}{c}[\models A,\ \models B]\\ \vdots \\ \models R\end{array}}{\models R}$$

and their counterparts and-valid-in-transition, or-validity, or-valid-in-transition and the suitable not-elimination rules.

Unfortunately, the rules handling the implication are only in parts elegantly:

$$\frac{\models A \quad \models A \,\mathsf{implies}\, B}{\models B} \qquad \frac{\models B}{\models A \,\mathsf{implies}\, B} \qquad \frac{\forall st \bullet \begin{array}{c}[st \models \mathsf{not}(\mathsf{is\_def}(A))]\\ \vdots \\ st \models B\end{array} \qquad \forall st \bullet \begin{array}{c}[st \models A]\\ \vdots \\ st \models B\end{array}}{\models A \,\mathsf{implies}\, B}$$

The problem is the implication introduction rule to the right that combines the two validity levels of the natural deduction calculus.

Undefinedness leads to an own side-calculus in OCL: Since from $\models A \,\mathsf{and}\, B$ we can conclude definedness both for $A$ and $B$ in all contexts, and since from $\models E \triangleq E'$ we can conclude that any subexpression in $E$ and $E'$ is defined (due to strictness to all operations in the expression language of OCL), a lot of definedness information is usually hidden in an OCL formula, let it be a method precondition or an invariant. In order to *efficiently* reason over OCL specification, it may be necessary precompute this information.

At present, we have only developed a simple setup for *fast_tac* according to the rules described above, which is already quite powerful, but not complete.

### 4.2 Rewriting

Rewriting OCL-formulae seems to have a number of advantages; mainly, it allows for remaining on the level of absolute validity which is easier to interpret, and it allows to hide the definedness concerns miraculously inside the equational calculus. The nastiness of the implication introduction can be shifted a bit further inside in an equational calculus: The two rules

$$A \,\mathsf{implies}(B \,\mathsf{implies}\, C) = (A \,\mathsf{and}\, B) \,\mathsf{implies}\, C$$
$$A \,\mathsf{implies}(B \,\mathsf{or}\, C) = (A \,\mathsf{implies}\, B) \,\mathsf{or}(A \,\mathsf{implies}\, C)$$

hold for all cases which — together with the other lattice rules for the logic — motivates a Hilbert-Style calculus for OCL; unfortunately, the assumption rule $A \,\mathsf{implies}\, A$ holds only if $A$ is defined in all contexts. At least, this gives rise to proof procedures that defer definedness reasoning to local places in a formula.

A useful mechanism to transport definedness information throughout an OCL formula can be based on Isabelle's simplifier that can cope with a particular type of rules. Derived congruence rewriting rules for *HOL-OCL* look like:

$$\frac{\begin{array}{c}[st \models A \vee st \models \mathsf{not}\,\mathsf{is\_def}(A)]\\ \vdots \\ B\ st = B\ st'\end{array}}{(A \,\mathsf{and}\, B)\ st = (A \,\mathsf{and}\, B')\ st} \qquad \frac{A\ st = A'\ st \quad B\ st = B'\ st}{(A \,\mathsf{or}\, B)\ st = (A' \,\mathsf{or}\, B')\ st}$$

allow for replacing, for example, variables occurrences by $\bot_{\mathscr{L}}$ or $\mathsf{true}$ if their undefinedness or validity follows somewhere in the context.

We discovered a further interesting technique for proving the equality of two formulae $P\ X$ and $Q\ X$ based on the idea to perform a case split by substituting $X$ by $\bot_{\mathscr{L}}$, $\mathsf{true}$ or $\mathsf{false}$. Unfortunately, it turns out that the rule:

$$\frac{P\ \bot_{\mathscr{L}} = P'\ \bot_{\mathscr{L}} \qquad P\ \mathsf{true} = P'\ \mathsf{true} \qquad P\ \mathsf{false} = P'\ \mathsf{false}}{P\ X = P'\ X}$$

is simply unsound due to the fact that it does not hold for *all* functions $P\ X\ c$, only if $P$ and $P'$ are state passing, which represents an invariant of our embedding (see Sec. 3.2). Fortunately, for all logical operators, state passing rules such as:

$$\frac{\mathrm{pass}\ P}{\mathrm{pass}(\lambda X \bullet \mathsf{not}(P\ X))} \qquad \frac{\mathrm{pass}\ P \quad \mathrm{pass}\ P'}{\mathrm{pass}(\lambda X \bullet (P\ X)\,\mathsf{and}(P'\ X))}$$

hold. Moreover, any function constructed by a lifting (and these are all library function definitions) are state passing. This allows for proof procedures built on systematic case distinctions which turn out to be efficient and useful.

The situation is similar for reasoning over strong and weak equality. For strong equality, we have *nearly* the usual rules of an equational theory with reflexivity, symmetry and transitivity. For Leibniz rule (substitutivity), however, we need again that the context $P$ is state passing:

$$\frac{\models a \triangleq b \quad \models P\ a \quad \mathrm{pass}\ P}{\models P\ b}$$

This is similar for strict equality, except for additional definedness constraints.

### 4.3 Lifting Theorems from HOL to the *HOL-OCL* Level

Since all operations in the library are defined extremely canonically by a combination of (optional) smashing, strictification and lifting operators, it is possible to derive automatically from generic theorems such as strictness rules, definedness propagation etc.:

$$\begin{aligned}
\mathrm{lift}_1(\mathrm{strictify}\ f)\ \bot_{\mathscr{L}} &= \bot_{\mathscr{L}} \\
\mathrm{lift}_2(\mathrm{strictify}(\lambda x \bullet \mathrm{strictify}(f\ x)))\ \bot_{\mathscr{L}}\ X &= \bot_{\mathscr{L}} \\
\mathrm{lift}_2(\mathrm{strictify}(\lambda x \bullet \mathrm{strictify}(f\ x)))\ X\ \bot_{\mathscr{L}} &= \bot_{\mathscr{L}} \\
\mathrm{is_def}(\mathrm{lift}_1(\mathrm{strictify}(\lambda x \bullet \mathrm{lift}(f\ x)))\ X) &= \mathrm{is_def}(X) \\
(\forall x\ y \bullet f\ x\ y = f\ y\ x) \Rightarrow \mathrm{lift}_2\ f\ X\ Y &= \mathrm{lift}_2\ f\ Y\ X
\end{aligned}$$

The last rule is used to lift a commutativity property from the HOL level to the *HOL-OCL*-level. With such lifting theorems, many standard properties were proven automatically in the library.

## 5 Application: Test Case Generation

A prominent example for automatic test case generation is the triangle problem [13]: Given three integers representing the lengths of the sides of a triangle, a small algorithm has to check, whether these integers describe invalid input or an equilateral, isosceles, or scalene triangle. Assuming a class Triangle with the operations isTriangle() (test if the input describes a triangle) and triangle() (classify the valid triangle) leads to the following OCL specification:

```
context Triangle :: isTriangle(s0, s1, s2: Integer): Boolean
pre:  (s0 > 0) and (s1 > 0) and (s2 > 0)
post: result =  (s2 < (s0 + s1)) and  (s0 < (s1 + s2))
                and  (s1 < (s0 + s2))

context Triangle :: triangle(s0, s1, s2: Integer): TriangType
pre:  (s0 > 0) and (s1 > 0) and (s2 > 0)
post: result = if (isTriangle(s0,s1,s2)) then
                   if (s0 = s1) then
                       if (s1 = s2) then
                           Equilateral::TriangType
                       else
                           Isosceles::TriangType
                       endif
                   else
                       if (s1 = s2) then
                           Isosceles::TriangType
                       else
                           if (s0 = s2) then
                               Isosceles::TriangType
                           else
                               Scalene::TriangType
                           endif
                       endif
                   endif
               else
                   Invalid::TriangType
               endif
```

Transforming this specification into *HOL-OCL*[3] leads to the following specification *triangle_spec* of the operation triangle():

$$\text{triangle\_spec} \equiv \lambda\, result\ s_1\ s_2\ s_3 \bullet result \triangleq (\text{if isTriangle } s_1\ s_2\ s_3 \text{ then if } s_0 \triangleq s_1 \text{ then if } s_1 \triangleq s_2 \text{ then equilateral else isosceles endif else if } s_1 \triangleq s_2 \text{ then isosceles else if } s_0 \triangleq s_2 \text{ then isosceles else scalene endif endif endif else invalid endif})$$

[3] In the following, we omit the specification of isTriangle().

For the actual test-case generation, we define triangle, which selects via Hilbert's epsilon operator (@) an eligible "implementation" fulfilling our specification:

$$\text{triangle} : [Integer_\alpha, Integer_\alpha, Integer_\alpha] \Rightarrow \text{Triangle}_\perp$$
$$\text{triangle } s_0\ s_1\ s_2 \equiv @result\bullet \models \text{triangle\_spec } result\ s_0\ s_1\ s_2$$

We follow the approach presented in [14] using a disjunctive normal form (DNF) for partition analysis of the specification and as a basis for the test case generation. In our setting this leads to the following main steps:

1. Eliminate logical operators except and, or, and not.
2. Convert the formula into DNF.
3. Eliminate unsatisfiable disjoints by using concurrence rewriting.
4. Select the actual set of test-cases.

Intermediate results are formulae with over 50 disjoints. The logical simplification can only eliminate simple logical falsifications, but this representation can tremendously be simplified by using *congruence rewriting*. Based on its deeper knowledge of the used data types (taking advantage of e.g. $\not\models$ isosceles $\triangleq$ invalid) this step eliminates many unsatisfiable disjoints caused by conflicting constraints. After the congruence rewriting, only six cases are left, respectively one for invalid inputs and one for equilateral triangles, and three cases describing the possibilities for isosceles triangles.

$$\begin{aligned}
&\text{triangle } s_0\ s_1\ s_2 = @result\bullet \models result \triangleq \text{invalid and not isTriangle s0 s1 s2}\\
&\text{or } result \triangleq \text{equilateral and isTriangle } s_0\ s_1\ s_2 \text{ and } s_0 \triangleq s_1 \text{ and } s_1 \triangleq s_2\\
&\text{or } result \triangleq \text{isosceles and isTriangle } s_0\ s_1\ s_2 \text{ and } s_0 \triangleq s_1 \text{ and } s_1 \not\triangleq s_2\\
&\text{or } result \triangleq \text{isosceles and isTriangle } s_0\ s_1\ s_2 \text{ and } s_0 \triangleq s_2 \text{ and } s_0 \not\triangleq s_1\\
&\text{or } result \triangleq \text{isosceles and isTriangle } s_0\ s_1\ s_2 \text{ and } s_1 \triangleq s_2 \text{ and } s_0 \not\triangleq s_1\\
&\text{or } result \triangleq \text{scalene and isTriangle } s_0\ s_1\ s_2 \text{ and } s_0 \not\triangleq s_1 \text{ and } s_0 \not\triangleq s_2 \text{ and } s_1 \not\triangleq s_2
\end{aligned}$$

These six disjoints represent the partitions, from which test cases can be selected, possible exploiting boundary cases like minimal or maximum Integers of the underlying implementation.

## 6 Conclusion

### 6.1 Achievements

We have presented a new formal semantic model of OCL in form of a conservative embedding into Isabelle/HOL that can cope with the requirements and the examples of the OCL standard 1.4. On the basis of the embedding, we derived several calculi and proof techniques for OCL. Since "deriving" means that we proved all rules with Isabelle, we can guarantee both the consistency of the semantics as well as the soundness of the calculi. Our semantics is organized in

a modular way such that it can be used to study the interdependence of certain language features (method recursion, executability, strictness, smashing, flattening etc.) which might be useful in the current standardization process of OCL. We have shown the potential for semantic based tools for OCL using automated reasoning by an exemplary test-case generation.

### 6.2 Related Work

Previous semantic definitions of OCL [15, 16, 5] are based on "mathematical notation" in the style of "naive set theory", which is in our view quite inadequate to cover so subtle subjects such as inheritance. Moreover, the development of proof calculi and automated deduction for OCL has not been in the focus of interest so far.

In [15], a formal operational semantics together with a formal type system for OCL 1.4 was presented. The authors focus on the issue of subject reduction, but do not define the semantic function for expressions whose evaluation may diverges. In [17], it is claimed that a similar OCL semantics is Turing complete. In contrast, our version of OCL admits an infinite state which turns allInstances into an unbounded universal quantifier; when adding least-fixpoint semantics for recursive methods (as we opt for), we are definitively in the world of non-executable languages.

Using a shallow embedding for an object oriented language is still a challenge. While the basic concepts in our approach of representing subtyping by the subsumption relation on polymorphic types is not new (c.f. for example [18, 19]), we have included concepts such as undefinedness, mutual recursion between object instances, dynamic types, recursive method invocation and extensible class hierarchies that pushes the limits of the approach a bit further.

### 6.3 Future Work

Beyond the usual sigh that the existing library is not developed enough (this type of deficiency is usually resolved after the first larger verification project in an embedding), we see the following extensions of our work:

- While our *fast_tac*-based proof procedure for OCL logic is already quite powerful, it is neither efficient nor complete (but should be for a fragment corresponding to propositional logic extended by definedness). More research is necessary (multivalued logics [20], Decision Diagrams).
- Since *HOL-OCL* is intended to be used over several stages of a software development cycle, a refinement calculus that formally supports this activity may be of particular relevance.
- Combining *HOL-OCL* with a Hoare-Logic such as $\mu$Java[8] can pave the way for an integrated formal reasoning over specifications and code.

## References


[1] OMG: Unified Modeling Language Specification (Version 1.4). (2001)
[2] Kobryn, C.: Will UML 2.0 be agile or awkward? CACM **45** (2002) 107–110
[3] OMG: Object Constraint Language Specification. [1] chapter 6
[4] Warmer, J., Kleppe, A.: The Object Contraint Language: Precise Modelling with UML. Addison-Wesley Longman, Reading, USA (1999)
[5] Warmer, J., Kleppe, A., Clark, T., Ivner, A., Högström, J., Gogolla, M., Richters, M., Hussmann, H., Zschaler, S., Johnston, S., Frankel, D.S., Bock, C.: Response to the UML 2.0 OCL RfP. Technical report (2001)
[6] Spivey, J.M.: The Z Notation: A Reference Manual. Prentice Hall (1992)
[7] Jones, C.B.: Systematic Software Development Using VDM. Prentice Hall (1990)
[8] Nipkow, T., von Oheimb, D., Pusch, C.: $\mu$Java: Embedding a programming language in a theorem prover. In Bauer, F.L., Steinbrüggen, R., eds.: Foundations of Secure Computation. Volume 175 of NATO Science Series F: Computer and Systems Sciences., IOS Press (2000) 117–144
[9] Brucker, A.D., Wolff, B.: A note on design decisions of a formalization of the OCL. Technical Report 168, Albert-Ludwigs-Universität Freiburg (2002)
[10] Broy, M., Facchi, C., Grosu, R., Hettler, R., Hussmann, H., Nazareth, D., Slotosch, O., Regensburger, F., Stølen, K.: The requirement and design specification language Spectrum, an informal introduction (V 1.0). Technical Report TUM-I9312, TU München (1993)
[11] Winskel, G.: The Formal Semantics of Programming Languages. MIT Press, Cambridge (1993)
[12] Cook, S., Kleppe, A., Mitchell, R., Rumpe, B., Warmer, J., Wills, A.: The Amsterdam Manifesto on OCL. Technical Report TUM-I9925, TU München (1999)
[13] North, N.D.: Automatic test generation for the triangle problem. Technical Report DITC 161/90, National Physical Laboratory, Teddington (1990)
[14] Dick, J., Faivre, A.: Automating the generation and sequencing of test cases from model-based specications. In Woodcock, J., Larsen, P., eds.: FME 93: Industrial-Strength Formal Methods. Volume 670 of LNCS., Springer (1993) 268–284
[15] Mandel, L., Cengarle, M.V.: A formal semantics for OCL 1.4. In M. Gogolla, C.K., ed.: UML 2001: The Unified Modeling Language. Modeling Languages, Concepts, and Tools. Volume 2185 of LNCS., Toronto, Springer (2001)
[16] Richters, M., Gogolla, M.: On Formalizing the UML Object Constraint Language OCL. In Ling, T.W., Ram, S., Lee, M.L., eds.: Proc. 17th Int. Conf. Conceptual Modeling (ER'98). Volume 1507 of LNCS., Springer (1998) 449–464
[17] Mandel, L., Cengarle, M.V.: On the expressive power of OCL. FM'99 (1999)
[18] Santen, T.: A Mechanized Logical Model of Z and Object-Oriented Specification. PhD thesis, Technical University Berlin (1999)
[19] Naraschewski, W., Wenzel, M.: Object-oriented verification based on record subtyping in Higher-Order Logic. In Grundy, J., Newey, M., eds.: Theorem Proving in Higher Order Logics. Volume 1479 of LNCS., Springer (1998) 349–366
[20] Hähnle, R.: Automated Deduction in Multiple-valued Logics. Oxford University Press (1994)