HOL-OCL: Experiences, Consequences and Design Choices

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 Based on experiences gained from an embedding of the Object Constraint Language (OCL) in higher-order logic [1], we explore several key issues of the design of a formal semantics of the OCL. These issues comprise the question of the interpretation of invariants, pre- and postconditions, an executable sub-language and the possibilities of refinement notions. A particular emphasize is put on the issue of mechanized deduction in UML/OCL specification.
Keywords: OCL, formal semantics, constraint languages, refinement

1 Introduction

The Object Constraint Language (OCL) [2, 3, 4] is part of the UML, the de-facto standard of object-oriented modeling. Being in the tradition of data-oriented formal specification languages like Z [5] or VDM [6], OCL is designed to make UML diagrams more expressive. In short, OCL is a three-valued Kleene logic with equality that allows for specifying constraints on graphs of object instances.

There is a need for both researchers and CASE tool developers[1] 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 started to provide a formal semantics in form of a conservative embedding of OCL into Isabelle/HOL which is described in [1]. In contrast to traditional paper-and-pencil-work in defining the semantics of a language, a theorem prover based formalization inside a powerful logical language such as higher-order logic (HOL) [7, 8] offers a number of advantages: First of all, the consistency of an embedded logic, if based on conservative extensions, can be guaranteed. Second, as already pointed out in [9], the use of a theorem prover works as *Occam's razor* in a formalization since machine-checked proofs enforce "a no-frills approach and often leads to unexpected simplifications". Third, incremental changes of a semantic theory can be facilitated since re-running the proof-scripts immediately reveal new problems in previous proofs, "thus freeing you from the tedium of having to go through all the details again". Further, programmable theorem provers such as Isabelle can

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

be used to study symbolic evaluation and thus prototypical tool development. For all these reasons, the embedding approach has been successfully applied for a number of "real" languages such as Java [10, 9], ML [11], CSP [12],or Z [13].

In this document, we summarize experiences and consequences of our formalization, including answers on questions raised in previous research on the precise meaning of OCL [14, 15, 4]: the issue of partial and total correctness, the role of exceptions, the precise meaning of invariants, the option for recursive query methods and executability of OCL, and its potential for formal refinement.

We proceed as follows: First, we will outline the underlying logical, methodological and technical framework. Second, we will present an abstraction of the key-concepts of our embedding [1]. On this basis, we enter in a discussion on the "research issues on OCL" as listed above and their answer in the light of our semantic model of OCL. Finally, we discuss our first experiences with mechanized deduction in UML/OCL specifications.

2 Formal and Technical Background

In this section we will outline the underlying logical, methodological and technical framework of our embedding of OCL into Isabelle/HOL.

2.1 Higher-order logic — HOL

Higher-order logic (HOL) [7, 8] is a classical logic with equality enriched by total polymorphic[2] higher-order functions. It is more expressive than first-order logic, since e.g. induction schemes can be expressed inside the logic. Pragmatically, HOL can be viewed as a combination of a typed functional programming language like Standard ML or Haskell extended by logical quantifiers. Thus, it often allows a very natural way of specification.

When extending logics, two approaches can be distinguished: the *axiomatic method* on the one hand and *conservative extensions* on the other. Extending the HOL core via axioms, i.e. introducing new, *unproven* laws seems to be the easier approach but it usually leads easily to inconsistency; given the fact that in any major theorem proving system the core theories and libraries contain several thousand theorems and lemmas, the axiomatic approach is worthless in practice. In contrast, a conservative extension introduces new constants (via *constant definitions*) and types (*type definitions*) only via a particular schema of axioms; the (meta-level) proof that axioms of this schema preserve consistency can be found in [16]. For example, a constant definition introduces a "fresh" constant symbol and a non-recursive equality axiom with the new constant at the left hand side, while the right-hand side is a closed expression. Thus, the new constant can be viewed as an abbreviation of previously defined constructs.

The HOL library provides conservative theories for the HOL-core based on type *bool*, for the numbers such as *nat* and *int*, for typed set theory based on

[2] to be more specific: *parametric polymorphism*

τ *set* and a list theory based on τ *list*. In this sense, the HOL type system is quite similar to the OCL type system, but offers no subtyping. Moreover, there are products, maps, and even a theory on real numbers and non-standard analysis.

2.2 Isabelle

Isabelle [17] is a *generic* theorem prover. New object logic's can be introduced by specifying their syntax and inference rules. Among other logics, Isabelle supports first order logic (constructive and classical), Zermelo-Fränkel set theory and HOL, which we choose as framework for *HOL-OCL*.

Isabelle consists of a logical engine encapsulated in an abstract data type *thm* in Standard ML; any *thm* object has been constructed by trusted elementary rules in the kernel. Thus Isabelle supports user-programmable extensions in a logically safe way. A number of generic proof procedures (*tactics*) have been developed; namely a simplifier based on higher-order rewriting and proof-search procedures based on higher-order resolution.

2.3 Shallow Embeddings vs. Deep Embeddings

We are now concerned with the question *how* a language is represented in a logic. In general, two techniques are distinguished: *shallow* and *deep* embeddings.

Deep embeddings represent the abstract syntax as a datatype and define a semantic function I from syntax to semantics.

Shallow embeddings define the semantics directly; each construct is represented by some function on a semantic domain.

Assume we want to embed the boolean OCL$_{light}$ operators[3] and and or into HOL. Note, that the semantics $I : expr \to env \to bool$ is a function that maps OCL expressions and environments to *bool*, where *environments* $env = var \to bool$ maps variables to *bool* values. Using a shallow embedding, we define directly:

$$x \,\mathsf{and}\, y \equiv \lambda\, e \bullet x\, e \wedge y\, e \qquad x \,\mathsf{or}\, y \equiv \lambda\, e \bullet x\, e \vee y\, e$$

Shallow embedding allows for direct definitions in terms of semantic domains and operations on them. In contrast, in a deep embedding, we have to define the syntax of OCL$_{light}$ as recursive datatype:

$$expr = \mathsf{var}\; var \;\mid\; expr \,\mathsf{and}\, expr \;\mid\; expr \,\mathsf{or}\, expr$$

and the explicit semantic function I:

$$\begin{aligned} I[\![\mathsf{var}\; x]\!] &= \lambda\, e \bullet e(x) \\ I[\![x \,\mathsf{and}\, y]\!] &= \lambda\, e \bullet I[\![x]\!]\, e \wedge I[\![y]\!]\, e \\ I[\![x \,\mathsf{or}\, y]\!] &= \lambda\, e \bullet I[\![x]\!]\, e \vee I[\![y]\!]\, e \end{aligned}$$

[3] Here we simplify the OCL semantics, by ignoring undefinedness and states.

This example reveals the main drawback of deep embeddings: the language is more distant to the underlying meta language HOL, i.e. semantic functions represent obstacles for deduction. The explicit syntax of deep embeddings enables induction proofs, however; for some meta-theoretic analysis, this may have advantages. Since we are interested in a concise semantic description of OCL and prototypical proof support, but not meta-theory, we chose a shallow embedding.

3 An Abstract Model of *HOL-OCL*

In this section, we will motivate some design goals of *HOL-OCL* (as described in [1]), namely extendibility and modular organization. Here, we are less concerned with an integrated solution of the technical problems, but with the underlying *concepts* of our approach.

3.1 Design Goals of *HOL-OCL*

A conservative shallow embedding enforces any entity of the language to have a *type* in HOL. This seems to be tedious at first sight, but has the foundational advantage that statements such as $x \in S$ never run into Russel's paradox; in a typed set theory, well-typedness assures consistency. The set theoretic foundation makes the treatment of what we call a *general semantics* of OCL quite difficult; such a semantics should interpret an OCL-expression not only in one class diagram, but also "the set of all possible extensions". Unfortunately, the semantic domain of general semantics requires concepts such as "the set of all universes over all class diagrams" (where a *universe* $\mathscr{U}$ is the set of all objects of a class diagram). Such a set is too large to be represented in a typed set theory and a critical construction in Zermelo-Fränkel set theory. Therefore, previous formal semantics to OCL ([14, 15, 4]) has been based on a "parameterized semantics" approach, i.e. the semantic function is parameterized by an arbitrary, but fixed diagram $\mathcal{C}$, and its definition is based on naive set theoretic reasoning [18].

Even if one is not too much concerned about the foundational problems with this approach, we argue that parametric semantics does not cover the most important aspect of object-orientation: *reuseability*. In principle, if one extends a class diagram $\mathcal{C}$ by a *diagram extension* $\mathcal{E}$ (containing new classes and inheritance relations) to a class diagram $\mathcal{C} \oplus \mathcal{E}$, one wants to reuse "everything" done for $\mathcal{C}$. In a theorem prover, the reuse of derived rules and theorems from the library or a user model is pragmatically vital. However, in the parameterized semantics approach, values in the two semantic functions $I_{\mathcal{C}}[\![e]\!]$ and $I_{\mathcal{C}\oplus\mathcal{E}}[\![e]\!]$ are unrelated, and, thus, theorems proven over the first do not carry over to the latter. In order to obtain proof-reuseability, we based *HOL-OCL* on the *extendible semantics* approach, that lies in between the parameterized and the general semantics. The idea is to introduce *extension variables* $\mathcal{X}$ and to represent diagrams by $\mathcal{C} \oplus \mathcal{X}$, such that diagram $\mathcal{C} \oplus \mathcal{E} \oplus \mathcal{X}'$ becomes an instance of $\mathcal{C} \oplus \mathcal{X}$. Since extension variables can be represented via type variables in HOL, values within a family of "diagrams extensions" become semantically related.

Representing a language that is still under construction (as OCL) is risky: if the various key features were changed, a lot of work may be lost. Keeping a semantic theory flexible is therefore a major challenge for the theorem-prover based analysis of language designs. We answered the problem by organizing the semantic definition for any type and any operator of OCL into layers, whose "semantical essence" is captured by certain combinators (i.e. type constructors for types and higher-order functions for operators). When embedding the types from HOL-library theories like *bool*, *string*, *set* etc, they all have to be "lifted" by adding a $\bot$-element denoting undefinedness:

$$Bool = bool \uplus \{\bot\} = bool_\bot$$

where $\uplus$ denotes the disjoint sum. When embedding operations, fundamental semantic principles like "all operations are strict" can be incorporated "once and for all" by defining a combinator "strictify", that is used in all definitions of OCL-operators (with some exceptions listed in the OCL standard). For instance, the strictified logical *not* can be defined as follows:

$$not' \equiv \lfloor\_\rfloor \circ (\text{strictify}(\neg\ ))$$

where $\lfloor\_\rfloor$ takes a value and embeds it into its disjoint sum with $\bot$, $\_\circ\_$ is the function composition, and $\neg\ \_$ the logical *not* in HOL on *bool.* The combinator is defined by:

$$\text{strictify}(f)(x) \equiv \begin{cases} \bot & \text{if } x = \bot \\ f\ z & \text{if } x = \lfloor z \rfloor. \end{cases}$$

Similarly, we define combinators that realize uniformly the semantic construction for state transitions (the actual definition for *not* is $\text{lift}_1(not')$, see below), for the "smashing" of collection types (see below) or for the treatment of *late binding* or *static binding* in recursive query methods.

The purpose of the following subsections is to introduce an "abstract version" of *HOL-OCL* that covers the most essential concepts but abstracts away the quite involved details with respect to the construction of objects in a state, with respect to types and with respect to parametric polymorphism, that implements these concepts. The details can be found in [1].

3.2 Foundation

In the following, we assume the base types *Boolean*, *String*, *Int*, etc. as lifted versions $bool_\bot$, $string_\bot$, etc. of the HOL library types, and $\mathcal{C}$, $\mathcal{C}'$,$\mathcal{C}''$ are denoting UML class diagrams. Further we assume an operation $\oplus$ and the class diagram extensions $\mathcal{E}$, $\mathcal{E}'$,$\mathcal{E}''$, universes $\mathcal{U}_\mathcal{C}$ of objects over a class diagram $\mathcal{C}$ (the universe also contains all values of the base types, states[4] $State(\mathcal{U}_\mathcal{C})$ containing objects of $\mathcal{U}_\mathcal{C}$), and $ST_\mathcal{C} \equiv State(\mathcal{U}_\mathcal{C}) \times State(\mathcal{U}_\mathcal{C})$.

[4] also called *object configurations* or *snapshots* in the literature

3.3 Methods in OCL

One vital motivation for OCL is its possibility to specify methods of class diagrams with pre- and postconditions:

```
context C.op(x:T1): T2
pre:  PRE
post: POST
```

where PRE and POST may be OCL expressions of type *Boolean*. PRE may contain at most the free variables *self* and x and accesses via path-expressions to an underlying "old" state $\sigma : \mathit{State}(\mathscr{U}_{\mathfrak{C}})$, while POST may additionally contain the free variable *result* and accesses to the "new" state $\sigma' : \mathit{State}(\mathscr{U}_{\mathfrak{C}})$.

The definition in the OCL 2.0 proposal is similar to Z operation schemas:

$$\begin{array}{|l}
\hline
\multicolumn{1}{l}{op} \\
\Delta \mathit{State} \\
x? : T_1 \\
\mathit{result}! : T_2 \\
\hline
\mathit{PRE}(x?, \sigma) \\
\mathit{POST}(x?, \mathit{result}!, \sigma, \sigma') \\
\hline
\end{array}$$

where $\mathit{State} : \mathit{Exp}$ is a schema over state variables σ containing the system invariant. The schema notation is just a notational variant of the (typed) set:

$$\big\{x? : T_1, \mathit{result}! : T_2, \sigma : \mathit{State}(\mathscr{U}_{\mathfrak{C}}), \sigma' : \mathit{State}(\mathscr{U}_{\mathfrak{C}}) \big| \\ \mathit{PRE}(x?, \sigma) \wedge \mathit{POST}(x?, \mathit{result}!, \sigma, \sigma')\big\}$$

Here, we propose a slight deviation from the OCL 2.0 proposal and allow PRE and POST to be undefined. Since sets $\{x \mid P(x)\}$ are just isomorphic to characteristic functions $P : \tau \to \mathit{bool}$, it is natural to view the semantics of an operation *op* as a function

$$I_{\mathfrak{C}}[\![op]\!] : T_1 \to T_2 \to ST_{\mathfrak{C}} \to \mathit{Boolean}$$

where $T_1 \to T_2 \to ST_{\mathfrak{C}} \to \mathit{Boolean}$ is just a curried (isomorphic) version of $T_1 \times T_2 \times ST_{\mathfrak{C}} \to \mathit{Boolean}$. When turning the semantics into a shallow embedding, one more massaging step is necessary:

$$I^S_{\mathfrak{C}}[\![op]\!] : (ST_{\mathfrak{C}} \to T_1) \to (ST_{\mathfrak{C}} \to T_2) \to ST_{\mathfrak{C}} \to \mathit{Boolean}$$

or just:

$$I^S_{\mathfrak{C}}[\![op]\!] : V_{\mathfrak{C}}(T_1) \to V_{\mathfrak{C}}(T_2) \to V_{\mathfrak{C}}(\mathit{Boolean})$$

where the type constructor $V_{\mathfrak{C}}(T_1)$ is an abbreviation for $ST_{\mathfrak{C}} \to T_1$.

Note that this semantic function interprets a *specification* of *op* and thus constructs a kind of "three valued relation" between input and output state, i.e. a state may be clearly related to its successor, or may be clearly unrelated, or the relationship may be undefined (see discussion below). Our solution is very similar to the OCL 2.0 proposal; however, we allow the distinction between unrelated states and an unspecified relation between the states (see Sec. 4.4). Hence, when semantically interpreting a concrete *operation* $op : T_1 \to T_1$, the resulting type will be:

$$I^S_{\mathcal{C}}[\![op]\!] : V_{\mathcal{C}}(T_1) \to V_{\mathcal{C}}(T_2).$$

Thus, there is a clear conversion scheme from OCL-types to *HOL-OCL* types, which corresponds to a semantic construction we call *lifting* that we support by appropriate combinators. The combinator lift_1 (for "lift 1-ary HOL-function to shallow OCL interpretation"; see also previous section) is designed to hide the "plumbing" with respect to the state transitions. It is defined by:

$$\text{lift}_1(f)(x)(\sigma, \sigma') \equiv f(x(\sigma, \sigma'))$$

(the combinators for the 2-ary and 3-ary case are analogous).

3.4 Logic

We define the Boolean constants true, false and $\bot_{\mathscr{L}}$ (undefined) as constant functions, that yield the lifted HOL-value for undefinedness, truth or falsehood:

$$\begin{aligned} \bot_{\mathscr{L}} &: V_{\mathcal{C}}(\mathit{Boolean}) & \bot_{\mathscr{L}} &\equiv \lambda(\sigma,\sigma') \bullet \lfloor \bot \rfloor \\ \mathsf{true} &: V_{\mathcal{C}}(\mathit{Boolean}) & \mathsf{true} &\equiv \lambda(\sigma,\sigma') \bullet \lfloor \text{True} \rfloor \\ \mathsf{false} &: V_{\mathcal{C}}(\mathit{Boolean}) & \mathsf{false} &\equiv \lambda(\sigma,\sigma') \bullet \lfloor \text{False} \rfloor \end{aligned}$$

Following the previous section, the types for the logical operators are as follows:

$$\begin{aligned} \mathsf{not} &: V_{\mathcal{C}}(\mathit{Boolean}) \to V_{\mathcal{C}}(\mathit{Boolean}) \\ \mathsf{and} &: V_{\mathcal{C}}(\mathit{Boolean}) \to V_{\mathcal{C}}(\mathit{Boolean}) \to V_{\mathcal{C}}(\mathit{Boolean}) \\ \mathsf{or} &: V_{\mathcal{C}}(\mathit{Boolean}) \to V_{\mathcal{C}}(\mathit{Boolean}) \to V_{\mathcal{C}}(\mathit{Boolean}) \end{aligned}$$

The logical operators and the "if then else endif" are the only exception from the rule that the OCL operators have to be strict. They must be defined individually. The definitions following the OCL 2.0 proposal are straight-forward, e.g.:

$$(S \,\mathsf{and}\, T)(\sigma,\sigma') \equiv \begin{cases} \lfloor S \wedge T \rfloor & \text{if } S(\sigma,\sigma') \neq \bot \wedge T(\sigma,\sigma') \neq \bot \\ \mathsf{false} & \text{if } S(\sigma,\sigma') \neq \bot \wedge S(\sigma,\sigma') = \mathsf{false} \\ \mathsf{false} & \text{if } T(\sigma,\sigma') \neq \bot \wedge T(\sigma,\sigma') = \mathsf{false} \\ \bot_{\mathscr{L}} & \text{otherwise} \end{cases}$$

Note that due to extensionality of the HOL equality:

$$\forall\, x \bullet f(x) = g(x) \Rightarrow f = g$$

the states can be "hidden away". For example, instead of:

$$(\mathsf{true\,and\,false})(\sigma,\sigma') = \mathsf{false}(\sigma,\sigma')$$

we can write equivalently:

$$\mathsf{true}\,\mathsf{and}\,\mathsf{false} = \mathsf{false}$$

The proof in Isabelle is easily done by applying extensionality and Isabelle's proof procedure. The usual truth tables for Kleene logics were derived similarly:

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	$\bot_{\mathscr{L}}$
$\bot_{\mathscr{L}}$	$\bot_{\mathscr{L}}$	$\bot_{\mathscr{L}}$

It is not difficult to derive with Isabelle the laws of the surprisingly rich algebraic structure of Kleene logics: Both and and or enjoy not only the usual associativity, commutativity and idempotency laws, but also both distributives and de Morgan laws. The main drawback of a three-valued Kleene logic is that the axiom of the excluded middle ($a \vee \neg\, a = \mathit{true}$) and the usual laws on implication do not hold, which makes their handling in proof procedures non-standard.

Based on the logic, we define as suggested in the OCL 2.0 proposal ([4], *satisfaction of operation specifications* (see definition 5.34)) two notions of *validity* of an OCL formula: a formula P may be *valid* for all state transitions ($\models P$) or *valid for some transition* (σ, σ') ($(\sigma, \sigma') \models P$). We define validity by:

$$\models P \equiv P = \mathsf{true} \quad \text{or} \quad (\sigma, \sigma') \models P \equiv P(\sigma, \sigma') = \mathsf{true}(\sigma, \sigma')$$

respectively. Note that $P = \mathsf{false}$ is equivalent to $\models \mathsf{not}\, P$ and $P = \bot_{\mathscr{L}}$ is equivalent to $\models$ is_undef P where is_undef $P \equiv \lambda\, st. \lfloor P\; st \neq \bot \rfloor$. This also extends to the *validity for some transition.*

3.5 Layered Semantics for the Library

A great advantage of shallow embeddings consists in the possibility to define the bulk of library operations via a *theory morphism* from HOL library theories: all OCL operators were defined uniformly in layers via combinators over HOL-operations, enabling theorems like e.g. "'concatenation is associative" (several thousand of these folk-theorems exist in the HOL-library and represent the basis for effective machine assisted reasoning) to be synthesized from their HOL-counterparts automatically; this extends to the operations for Integer, Real and Strings. For the parametric collection types (Sets, Bags, Sequences, and Tuples (as proposed in [4]), two further layers have to be considered: *flattening* and *smashing.* While the OCL standard 1.4 requires an automated flattening (whose semantics is not formally defined), the proposal for OCL 2.0 introduces a "manual" flattening operator which is also our preference. Further, for collections the question arises how to handle the case of undefined values inserted into a collection. There are essentially two different possibilities for their treatment: Tuples, for instance, may be defined as follows:

$$(\bot, X) = (Y, \bot) = \bot$$

with the consequence:

$$\text{first}(X, \perp) = \perp \quad \text{and} \quad \text{second}(\perp, Y) = \perp$$

Alternatively, tuples may be defined as:

$$(\perp, X) \neq (Y, \perp) \neq \perp$$

with the natural consequence:

$$\text{first}(X, \perp) = X \quad \text{and} \quad \text{second}(\perp, Y) = Y$$

In [19, 20], the former is called "smashed product", while the latter is the standard product. We also apply this terminology for sets, bags and sequences and suggest the use of smashed collection types and strict access operations (the OCL 2.0 proposal suggests a non-strict, even non-executable includes operation). Smashed collections mirror the operational behavior of programming languages such as Java and, hence, pave the way for an executable OCL subset (see also Sec. 4.6).

4 Consequences

It is worth noting that OCL semantics is traditionally considered as a *denotational semantics* (see also [20, chapter 5]) or *state transition semantics* for imperative languages. We will discuss the consequences for the notion of *invariant*, *method specifications*, their *refinement* and the connection to Hoare-logics and to programming languages implementing method specifications.

4.1 On Invariants in OCL

In the light of *HOL-OCL*, we have no problem with the explanation:

Object Constraint Language Specification [2] (version 1.4), page 6-52

> An OCL expression is an invariant of the type and must be true for all instances of that type at any time.

that raised some criticism by some researchers (e.g. [21]). Since OCL semantics describes "state transitions", *at any time* means at any state that is reachable by the state transition relation. The issue of "intermediate states" that may violate invariants is a problem of a refinement notion or the combination with an axiomatic semantics (see Sec. 4.4).

Even general recursion based on fixed-points for query operations does not change the picture since query operations may not have side-effects.

4.2 On Method Specifications

In [21] it was asked what the precise nature of pre-and postconditions of a method is, and what happens if a precondition of a method m is not fulfilled. As possible behaviors, four cases are listed: the implementation of m might raise an exception, or may diverge, or may terminate without changing the state, or may produce a more or less arbitrary state. The authors argue that a pragmatically useful notion of specification should at least exclude the last case.

In our formalization (following OCL 2.0), both conditions are conjoint and form a *method specification.* Thus, a *false* precondition simply says that there is "no transition" from a state to its successor; this corresponds to the operational behavior of an exception, a divergence or a deadlock.

Moreover, an *undefined* precondition may either result in an undefined or false method specification; in the former case, no statement on the implementation is made, i.e. it may behave arbitrarily. Thus, when writing a specification, there is the possibility to explicitly distinguish these possibilities: a precondition $a.c > 5$ makes no statement for the case that a is an undefined object in a particular state (provided the postcondition is valid), in $\text{not}\,\text{is\_undef}(a)$ and $a.c > 5$, however, it is explicitly specified that there is no successor state, if a is not defined in the previous state. We believe that this distinction is sensible and useful in connection with requirement collection and with refinement, albeit we foresee that these subtleties will not always be easy to swallow by practitioners.

Further, [21] argues that there is the necessity for distinguishing total and partial correctness for OCL, and develop a formal machinery based on four correctness notions. Here is a confusion about the type of semantics: a *precondition* (or postcondition respectively) in the sense of Hoare-logics [20] is a "predicate over admissible states" (thus a set of states), that were related over a *command.* Hoare-logics describes the annotation of command sequences with predicates. In contrast, denotational OCL semantics just describes the relation on states (the keyword pre resp. post are purely syntactical). The difference can be seen best at one feature: in OCL, any variable that should remain invariant must be stated explicitly by $x = x@\text{pre}$ (here, the semantics should be changed in order to limit this effect, say, to the attributes of a class), otherwise arbitrary transitions are possible; in contrast, in Hoare-logics, not occurring variables remain constant as a result of the consequence rule [20, p. 89]. Therefore, these correctness notions simply do not apply for OCL, they apply for a refinement of OCL to code.

In [21], two "semantic styles" for method specifications were distinguished: the $PRE \wedge POST$-style and the $PRE \Rightarrow POST$ style. While we adhere to the former (following the OCL 2.0 proposal), the authors of [21] advocate the latter. The $PRE \wedge POST$-style based on Kleene logics allows for two types of independent requirement collections that correspond to conjunction and disjunction (similar to the usual pragmatics in the Z schema calculus [22]). Both types profit from a style of specifications that uses undefinedness whenever possible in order to avoid over-specification. Conversely, conjunctions and disjunctions of method specifications allow for straight-forward *splitting transformations* of method specifications along the usual distributivity and de Morgan laws.

4.3 On Undefinedness and the OCL Logics

The current OCL 2.0 proposal explicitly requires an explicit value for undefinedness and forces the logics to be a Kleene logic, prescribing algebraic laws like

$\bot_{\mathscr{L}}$ $\mathsf{and}\,\mathsf{false} = \mathsf{false}$ etc. This particular choice has raised criticism [21], both for methodological and tool-technological reasons (the latter will be discussed in Sec. 5). In particular, another implication is proposed:

$$A\,\mathsf{implies}\,B \equiv \mathsf{not}\,A\,\mathsf{or}(A\,\mathsf{and}\,B)$$

instead of the version prescribed in the standard:

$$A\,\mathsf{implies}\,B \equiv \mathsf{not}\,A\,\mathsf{or}\,B$$

A routine analysis with Isabelle reveals that both definitions enjoy the same laws for a Hilbert-Style calculus:

$$\begin{aligned} A\,\mathsf{implies}\,B\,\mathsf{implies}\,C &= A\,\mathsf{and}\,B\,\mathsf{implies}\,C \\ A\,\mathsf{implies}(B\,\mathsf{or}\,C) &= A\,\mathsf{implies}\,B\,\mathsf{or}\,A\,\mathsf{implies}\,C \\ A\,\mathsf{implies}\,A &= A \quad \text{(provided } A \neq \bot_{\mathscr{L}}) \end{aligned}$$

However, the first definition fulfills $\bot_{\mathscr{L}}\,\mathsf{implies}\,A = \bot_{\mathscr{L}}$, while the second fulfills $\bot_{\mathscr{L}}\,\mathsf{implies}\,\mathsf{true} = \mathsf{true}$. While we found the first version slightly more intuitive, we do not see any fundamental reasons for preferring one definition to the other. Also, we investigated both implication versions in the context of a natural deduction calculus for OCL; here, the differences were also minimal.

4.4 On Refinement in OCL

Based on denotational method semantics and based on the validity notion introduced in Sec. 3.4, it is now an easy exercise to adopt, for example, the standard data refinement notion of Z ([5, p. 138], c.f. [22], where it is called "forward simulation"). As a prerequisite, we introduce an abstraction

predicate R that relates two states which may even be of two different data universes, i.e. we may have an abstract operation based on $ST_{\mathfrak{C}}$, while the implementing operation is based on $ST_{\mathfrak{C}'}$. Further, we define the usual predicate transformer $pre_{s,in}(P)$ which decides if for a state s and an input in there exists a successor state and output that makes the method specification valid. Recall that this "semantic" precondition construction is necessary since the PRE and POST part of a specification are a purely syntactical separation.

Our first refinement notion — called *validity refinement* — is defined in HOL for the abstract method specification M_{abs} and the concete method specification M_{conc} of type $V_{\mathfrak{C}}(T_{in}) \to V_{\mathfrak{C}}(T_{out}) \to V_{\mathfrak{C}}(Boolean)$ as follows:

$$\begin{aligned} M_{abs} \sqsubseteq_V^R M_{conc} = {} & \forall\, s_a s_c in \bullet pre_{s_a,in}(M_{abs}) \wedge (s_a, s_c) \in R \Rightarrow pre_{s_c,in}(M_{conc}) \\ & \wedge \forall\, s_a s_c s'_c in\ out \bullet pre_{s_a,in}(M_{abs}) \wedge (s_a, s_c) \in R \\ & \quad \wedge (s_c, s'_c) \vDash M_{conc}\, in\ out \\ & \quad\quad \Rightarrow (\exists\, s'_a \bullet (s'_a, s'_c) \in R \wedge (s_a, s'_a) \vDash M_{abs}\, in\ out) \end{aligned}$$

Validity refinement is based on the idea, that whenever a transition is possible in M_{abs}, then it must be possible in a corresponding implementation transition in

M_{conc} to find a successor state s'_a which is related to a possible successor state of M_{abs}. This refinement notion is only based on valid transitions. In the presence of Kleene logic, we can distinguish undefined transitions from impossible ones by replacing validity by invalidity. We define "failure refinement" as follows:

$$M_{abs} \sqsubseteq_F^R M_{conc} = M_{abs} \sqsubseteq_V^R M_{conc} \wedge F(M_{abs}) \sqsubseteq_V^R F(M_{conc})$$

where $F\,M\,in\,out$ = not($M\,in\,out$) (Recall that $M\,in\,out$ = false is equivalent to $\models$ not($M\,in\,out$)). Thus, the implementation notions describe in different degrees of precision that the underlying transition systems were refined to "more deterministic and more defined ones".

Of course, other refinement notions may be transferred to OCL along the lines presented above; e.g. *backward simulation* [22]. Further, it is possible to link method specifications to implementations in concrete code of a programming language like Java. Technically, this is an integration of a denotational semantics with a Hoare logic (c.f. [20], see also the corresponding formal proofs in Isabelle [23]); meanwhile, it is feasible to combine *HOL-OCL* with a Hoare logic for NanoJava [9]. Of course, such a combination will inherently be programming language specific. An in-depth discussion of these alternatives, however, is out of the scope of this paper.

4.5 On Recursive Methods

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 definitions 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. $\perp_{\mathcal{L}}$. This is consistent with the *least fixpoint* in the theory of complete partial orderings (cpo; c.f. [20]). This idea has already been proposed by [24].

The theory of cpo's is a strict extension of semantic domains with undefinedness and yields a least-fixpoint operator, which gives semantics on recursive equations of methods. This enables *method implementations* that can be handled as an add-on to the OCL standard (see [1] for more details). Since recursive methods are executable and increase the expressive power of OCL, we encourage their use. However, this extension does not come for free: first, the semantics of a method invocation (static binding vs. late binding) must be clarified; Moreover, with respect to code generation, a particular treatment of undefinedness in the logical connectors is necessary.

4.6 On Executability of OCL

OCL is based on 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, operations are defined as *strict*, 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 in handy when generating code for assertions or when animating specifications.

However, seen from meta-language such as HOL, many of these restriction seem to be ad hoc add complexity, while excluding expressiveness. For example, states must be finite — this is a restriction never needed in the semantics; allInstances of basic types like *Int* is defined as $\bot_{\mathscr{L}}$, etc. Moreover, these restrictions tend to make the language "not self-contained", i.e. some side-conditions like "the operation must be associative" (see the definition of collection->sum():T in [2]) of OCL can not be treated inside OCL and make a meta-language such as HOL necessary for a complete formal treatment). Further, these restrictions represent an obstacle for defining a library of mathematical definitions and theorems comparable to the *Mathematical Toolkit* in Z, which turned out to be vital for its success. We suggest to omit all these restrictions and to define an executable sub-language of OCL, which could comprise recursive methods, strict versions of the logical operators, smashed collections, etc. Admitting infinite sets paves the way for lazy evaluation code schemes and their animation.

5 Mechanization

HOL-OCL provides a natural deduction calculus and a Hilbert-style calculus which builds together with powerful OCL rewriting rules the foundation for automated proof techniques for OCL.

The basis of our deduction calculus are the notions for validity introduced in Sec. 3.4. Since they are in itself two-valued judgments in HOL, there is no fundamental problem to reason about these in Isabelle's (two-valued) proof engine, let it be the tableaux-prover or the rewrite engine. In particular the latter can be used to compute normal forms of OCL specifications, i.e. disjunctive normal forms suitable to generate test cases out of a method specification (see [1] for more details).

Information about *definedness* is often "hidden" in the context of OCL formulae, e.g. when rewriting B in the context $\vDash A \,\mathsf{and}\, B$, we can assume $\vDash A$ and therefore the definedness of A. This kind of *context rewriting* can be used in Isabelle's rewriting procedure, that can therefore simplify:

```
context A pre: self.x <= 0  and self.x <= self.x
```

to

```
context A pre: self.x <= 0
```

Note, that $\models x <= x$ does not hold *in general.*

Whereas our experiments with *HOL-OCL* represent a proof-of-technology for automated reasoning, we cannot make any statements about its efficiency for large examples at present.

6 Conclusion

We have presented an abstraction of a concrete formalization project of OCL using a shallow, conservative embedding into Isabelle/HOL, called *HOL-OCL*, in order to make OCL more understandable and discussed pragmatical, methodological and tool-oriented consequences. *HOL-OCL* is designed to be as close as possible to the OCL 2.0 proposal. The construction of *HOL-OCL* can guarantee the consistency, delivers formally proven sound rules and provides a prototypical implementation framework for OCL tools.

Based on our discussion, we draw the following major consequences:

- Although we find Kleene logic sometimes quite non-intuitive, we do not see any evidence that it causes major problems with respect to methodology or tool support,
- Method specification semantics needs to be extended by mechanisms that enforces "most" of the state to remain equal (cf. Sec. 4.2),
- The underlying concepts of OCL make it compatible for recursion based on least-fixpoint semantics. This possible future extension requires some changes of the OCL 2.0 proposal (smashed collections, strict includes).
- We would encourage a simplification of the OCL semantics by admitting infinite sets and general quantifiers in order to make OCL as a logic more self-contained and more abstract.

Minor consequences discussed in Chapter 4 are summarized in Tab. 1. We compare the actual OCL standard [2], the OCL 2.0 proposal [4] and our suggestions:

	OCL 1.4 [2]	OCL 2.0 RfP [4]	*HOL-OCL* preference
extendible universes	☐	☐	☑
general recursion	☐	☐	☑
smashing	?	☐	☑
automated flattening	☑	☐	☐
tuples	☐	☑	☑
finite state	☑	☑	☐
general Quantifiers	☐	☐	☑
allInstances finite	☑	☑	☐
Kleene logic	☑	☑	☑
strong and weak equality	☐	☑	☑

☑: supported ☐: unsupported

Table 1. Comparison of important properties

References

[1] Brucker, A.D., Wolff, B.: A proposal for a formal OCL semantics in Isabelle/HOL. In Muñoz, C., Tahar, S., Carreño, V., eds.: TPHOLs 2002. LNCS. Springer (2002)
[2] OMG: Object Constraint Language Specification. (2001)
[3] Warmer, J., Kleppe, A.: The Object Contraint Language: Precise Modelling with UML. Addison-Wesley (1999)
[4] 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 (2002)
[5] Spivey, J.M.: The Z Notation: A Reference Manual. 2nd edn. Prentice Hall (1992)
[6] Jones, C.B.: Systematic Software Development Using VDM. Prentice Hall (1990)
[7] Church, A.: A formulation of the simple theory of types. Journal of Symbolic Logic **5** (1940) 56–68
[8] Andrews, P.B.: An Introduction to Mathematical Logic and Type Theory: To Truth Through Proof. Academic Press (1986)
[9] Oheimb, D.v., Nipkow, T.: Hoare logic for NanoJava: Auxiliary variables, side effects and virtual methods revisited. In Eriksson, L.H., Lindsay, P.A., eds.: Formal Methods – Getting IT Right (FME'02). LNCS 2391, Springer (2002) 89–105
[10] Nipkow, T., von Oheimb, D., Pusch, C.: μJava: 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
[11] Gunter, E.L., VanInwegen, M.: HOL-ML. In Joyce, J., Seger, C., eds.: Higher Order Logic Theorem Proving and Its Applications. LNCS 780, Springer (1994) 61–73
[12] Tej, H., Wolff, B.: A corrected failure-divergence model for CSP in Isabelle/HOL. In Fitzgerald, J., Jones, C., Lucas, P., eds.: FME 97. LNCS 1313, Springer (1997)
[13] Kolyang, Santen, T., Wolff, B.: A structure preserving encoding of Z in Isabelle/HOL. In von Wright, J., Grundy, J., Harrison, J., eds.: TPHOLs. LNCS 1125, Springer (1996)
[14] Mandel, L., Cengarle, M.V.: A formal semantics for OCL 1.4. In M. Gogolla, C.K., ed.: UML 2001. LNCS 2185, Springer (2001)
[15] Richters, M., Gogolla, M.: On Formalizing the UML Object Constraint Language OCL. (In: Conceptual Modeling (ER 1998)
[16] Gordon, M.J.C., Melham, T.F.: Introduction to HOL. Cambridge Press (1993)
[17] Paulson, L.C.: Isabelle: A generic theorem prover. LNCS 825. Springer (1994)
[18] Halmos, P.R.: Naive Set Theory. van Nostrand (1979)
[19] Mosses, P.D.: Denotational semantics. 1 edn. Elsevier (1990)
[20] Winskel, G.: The Formal Semantics of Programming Languages. MIT Press (1993)
[21] Hennicker, R., Hussmann, H., Bidoit, M.: On the precise meaning of OCL constraints. In Clark, T., J.Warmer, eds.: Advances in Object Modelling with the OCL. LNCS 2263, Springer (2002) 69–84
[22] Woodock, J., Davies, J.: Using Z. Prentice Hall (1996)
[23] `www.cl.cam.ac.uk/Research/HG/Isabelle/library/HOL/IMP/index.html`.
[24] Cook, S., Kleppe, A., Mitchell, R., Rumpe, B., Warmer, J., Wills, A.: The Amsterdam Manifesto on OCL. Technical Report TUM-I9925, TU München (1999)