Noname manuscript No.
(will be inserted by the editor)

Semantics, Calculi, and Analysis for Object-oriented Specifications

Achim D. Brucker · **Burkhart Wolff**

the date of receipt and acceptance should be inserted later

Abstract We present a formal semantics for an object-oriented specification language. The formal semantics is presented as a conservative shallow embedding in Isabelle/HOL and the language is oriented towards OCL formulae in the context of UML class diagrams. On this basis, we formally derive several equational and tableaux calculi, which form the basis of an integrated proof environment including automatic proof support and support for the analysis of this type of specifications.

We show applications of our proof environment to data refinement based on an adapted standard refinement notion. Thus, we provide an integrated formal method for refinement-based object-oriented development.

Keywords UML · OCL · object-oriented specification · refinement · formal methods

1 Introduction

The *Unified Modeling Language* (UML) [25] has been widely accepted throughout the software industry for modeling object-oriented software systems and is successfully applied to diverse domains [19]. UML is supported by major Computer Aided Software Engineering (CASE) tools and integrated into an object-oriented software development process model that stood the test of time. The *Object Constraint Language* (OCL) [24] is a textual extension of the core UML that allows for constraining UML models.

In research communities, UML/OCL has attracted interest for various reasons:

1. it is a formalism with a "programming language face," which is perhaps easier to adopt by software developers notoriously hostile to mathematical notation,

A.D. Brucker
SAP Research, Vincenz-Priessnitz-Str. 1, 76131 Karlsruhe, Germany
E-mail: achim.brucker@sap.com
URL: http://www.brucker.ch/

B. Wolff
Université Paris-Sud, Parc Club, 4, Rue Jaques Monod, 91893 Orsay Cedex, France
E-mail: wolff@lri.fr
URL: http://www.lri.fr/~wolff/

2. it puts forward the idea of an object-oriented specification formalism, turning objects and inheritance into the center of the modeling technique, and
3. it provides in many respects a "core language" for object-oriented modeling which makes it a good target for research of object-oriented semantics.

Here, item 1 refers not only to the concrete syntax (which we will largely ignore throughout this paper), but also to semantics: OCL semantics comprises the notion of undefinedness to model exceptional computations abstractly; this concept is deeply integrated into the logic and presents a particular challenge to deductive systems. Further, especially item 2 makes OCL rather different from logical languages such as first-order logic, higher-order logic, set theory and derived specification formalisms such as Z [35] or VDM [16]. Following a long platonic tradition in logic, these languages have a foundation in the notion of *values* and the definition of (hierarchies of) relations over them. In contrast, OCL allows for specifying constraints on the state consisting of object instances linked via references, i. e., its *object graphs*, and the transition relation over this state. This remarkably different perspective makes semantics for object-oriented specifications difficult. Comparing OCL with the two related approaches JML and Spec#, the main difference is that OCL attempts to abstract from concrete object-oriented programming languages, while JML and Spec# are designed as annotation-languages for a particular one.

In this paper, we present HOL-OCL, which is a *language*, based on the UML/OCL standard, as well as a *system* and a *methodology*. We will use HOL-OCL to illustrate our contributions, which we divide into the following categories:

1. We present a machine-checked formal semantics for *object-oriented data models* (e. g., class systems as present in the UML) comprising subtyping, single-inheritance, casting, dynamic and static types.
2. We present a formal, machine-checked semantics for UML and its assertion language OCL; Since both semantics are presented as a type-safe, shallow embedding, they are particularly suited as a basis for a proof environment.
3. Based on this semantic embedding, we derive several proof calculi consisting of derived rules constructed by machine-checked proofs, in particular equational and tableaux calculi.
4. We develop specific proof automation, both on the level of the data models, and on the level of the derived rules.
5. We present a method to analyze object-oriented data-models and combine it with a forward refinement method [31, 35]; thus, high-level specifications can be refined to executable ones.

Except from background presentation in Section 2, the plan of the paper follows the structure of our contribution list: Section 3 is devoted to the conservative embedding of the semantics assertions on them, Section 4 to the derivation of the proof calculi and automated proof-support, and Section 5 is concerned with the methodology of specification analysis and refinement.

2 Background

2.1 A Guided Tour Through UML/OCL

The Unified Modeling Language (UML) comprises a variety of model types for describing static (e. g., class models, object models) and dynamic (e. g., state-machines, activity graphs) system properties. One of the more prominent model types of the UML is the *class*

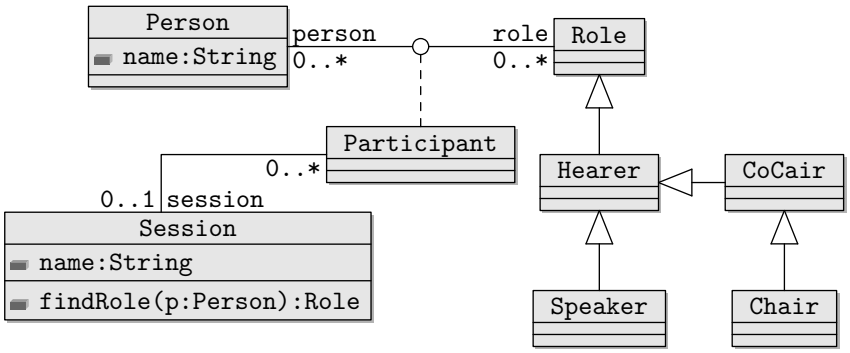


Figure 1 A simple UML class model representing a conference system for organizing conference sessions: persons can participate, in different roles, in a session.

model (visualized as *class diagram*) for modeling the underlying data model of a system in an object-oriented manner. As a running example, we model a part of a conference management system. Such a system usually supports the conference organizing process, e. g., creating a conference Website, reviewing submissions, registering attendees, organizing the different sessions and tracks, and indexing and producing the resulting proceedings. In this example, we constrain ourselves to the process of organizing conference sessions; Figure 1 shows the class model. We model the hierarchy of roles of our system as a hierarchy of classes (e. g., `Hearer`, `Speaker`, or `Chair`) using an *inheritance* relation (also called *generalization*). In particular, *inheritance* establishes a *subtyping* relationship, i. e., every `Speaker` (*subclass*) is also a `Hearer` (*superclass*).

A class does not only describe a set of *instances* (called *objects*), i. e., record-like data consisting of *attributes* such as `name` of class `Session`, but also *operations* defined over them. For example, for the class `Session`, representing a conference session, we model an operation `findRole(p:Person):Role` that should return the role of a `Person` in the context of a specific session; later, we will describe the behavior of this operation in more detail using OCL. In the following, the term object describes a (run-time) instance of a class or one of its subclasses.

Relations between classes (called *associations* in UML) can be represented in a class diagram by connecting lines, e. g., `Participant` and `Session` or `Person` and `Role`. Associations may be labelled by a particular constraint called *multiplicity*, e. g., `0..*` or `0..1`, which means that in a relation between participants and sessions, each `Participant` object is associated to at most one `Session` object, while each `Session` object may be associated to arbitrarily many `Participant` objects. Furthermore, associations may be labelled by projection functions like `person` and `role`; these implicit function definitions allow for OCL-expressions like `self.person`, where `self` is a variable of the class `Role`. The expression `self.person` denotes persons being related to the a specific object `self` of type role. A particular feature of the UML are *association classes* (`Participant` in our example) which represent a concrete tuple of the relation within a system state as an object; i. e., associations classes allow also for defining attributes and operations for such tuples. In a class diagram, association classes are represented by a dotted line connecting the class with the association. Associations classes can take part in other associations. Moreover, UML supports also *n*-ary associations (not shown in our example).

We refine this data model using the Object Constraint Language (OCL) for specifying additional invariants, preconditions and postconditions of operations. For example, we specify that objects of the class `Person` are uniquely determined by the value of the `name` attribute and that the attribute `name` is not equal to the empty string (`''`):

```
context Person
  inv: name ≠ '' ∧ Person::allInstances()->isUnique(p:Person | p.name)
```

Moreover, we specify that every session has exactly one chair by the following invariant (called `onlyOneChair`) of the class `Session`:

```
context Session
  inv onlyOneChair:
      self.participants->one( p:Participant | isType_Chair(p.role))
```

where $\text{isType}_{\texttt{Chair}}(\texttt{p.role})$ evaluates to true, if `p.role` is of *dynamic type* `Chair`. Besides the usual *static types* (i. e., the types inferred by a static type inference), objects in UML and other object-oriented languages have a second *dynamic* type concept. This is a consequence of a family of *casting functions* (written $o_{[C]}$ for an object o into another class type C) that allows for converting the static type of objects along the class hierarchy. The dynamic type of an object can be understood as its "initial static type" and is unchanged by casts (see Section 3 for details). We complete our example by describing the behavior of the operation `findRole` as follows:

```
context Session::findRole(person:Person):Role
  pre:  person ∈ self.participates.person
  post: result=self.participants->one(p:Participant |
                                 p.person ≐ person ).role
        ∧ self.participants ≐ self.participants@pre
        ∧ self.name ≐ self.name@pre
```

where in post-conditions, the operator `@pre` allows for accessing the previous state.

In UML, classes can contain attributes of the type of the defining class. Thus, UML can represent (mutually) recursive datatypes. Moreover, OCL introduces also recursively specified operations.

A key idea of defining the semantics of UML and extensions like SecureUML [8] is to translate the diagrammatic UML features into a combination of more elementary features of UML and OCL expressions [13]. For example, associations are usually represented by collection-valued class attributes together with OCL constraints expressing the multiplicity. Thus, having a semantics for a subset of UML and OCL is tantamount for the foundation of the entire method.

2.2 Formal and Technical Background of HOL in Isabelle

2.2.1 The Logical Framework Isabelle

Isabelle [23] is a logical framework providing a logical core language based on an intuitionistic fragment of higher-order logic (HOL). Isabelle is based on a typed version of the λ-calculus: types τ are defined as $\tau ::= \alpha \mid \chi(\tau,\ldots,\tau)$, where the set of *type variables* α is ranging over $\alpha,\beta,\ldots$, and where the set of *type constructors* χ contains the function space $\_ \Rightarrow \_$. Further, we use infix notation; e. g., instead of $\_ \Rightarrow \_(\tau_1,\tau_2)$ we write $\tau_1 \Rightarrow \tau_2$; multiple applications like $\tau_1 \Rightarrow (\ldots \Rightarrow (\tau_n \Rightarrow \tau_{n+1})\ldots)$ are also written as $[\tau_1,\ldots,\tau_n] \Rightarrow \tau_{n+1}$.

The terms of Isabelle are λ-terms defined as $\Lambda ::= C \mid V \mid \lambda V.\Lambda \mid \Lambda\Lambda$, where C is the set of *constants*, V is the set of *variables* like x, y, z. *Abstractions* and *applications* are written $\lambda x.\ e$ and $e\,e'$ or $e(e')$. A subset of λ-terms may be *typed*, i. e., terms may be associated to types by an inductive type inference system similar to the programming language Haskell. The built-in logical core language comprises a congruence $\_ \equiv \_$, a meta-implication $\_ \Longrightarrow \_$ and a meta-quantifier $\bigwedge x.Px$. The meta-implication helps to represent *logical rules*: a Horn-clause $A_1 \Longrightarrow \ldots \Longrightarrow A_n \Longrightarrow A_{n+1}$, written $[\![A_1;\ldots;A_n]\!] \Longrightarrow A_{n+1}$, is viewed as a rule of the form "from assumptions A_1 to A_n, infer conclusion A_{n+1}":

$$\frac{A_1 \quad \ldots \quad A_n}{A_{n+1}} \qquad \frac{\begin{array}{c}[A_1]\\ \vdots\\ A_2\end{array}}{A}\ . \tag{1}$$

The second rule to the right represents the natural deduction rule "if A_2 can be inferred from assumption A_1, infer A" which is represented in Isabelle as a rule of the form $(A_1 \Longrightarrow A_2) \Longrightarrow A$.

The meta-quantifier helps to represent eigenvariables and turns out to be a flexible mechanism to represent Skolemization for quantifiers; dually, Isabelle's term language comprises meta-variables (denoted $?x, ?y, ?z, \ldots$) that represent "terms to be substituted" during proof. For example, universal quantifiers are captured by the rules:

$$\frac{\forall x.\ P(x)}{P(?x)} \qquad \text{and} \qquad \frac{\bigwedge x.\ P(x)}{\forall x.\ P(x)}\ . \tag{2}$$

The deduction engine of Isabelle is based on higher-order resolution; this means that the meta-variables are substituted during the inferences as needed.

2.2.2 The Meta-language HOL

Classical higher-order logic (HOL) [11,4] is the instance of Isabelle which is mostly used and developed. A few axioms describe the logical core system based on the logical type bool with the logical connectives $\neg\_$, $\_ \wedge \_$, $\_ \vee \_$ and $\_ \to \_$ which are constants of type bool $\Rightarrow$ bool or $[\text{bool}, \text{bool}] \Rightarrow$ bool. Quantifiers are represented by higher-order abstract syntax; this means that $\forall\_.\ \_$ and $\exists\_.\ \_$ are usual constants of type $(\alpha \Rightarrow \text{bool}) \Rightarrow$ bool and that terms of the form $\forall(\lambda x.\ Px)$ are written $\forall x.\ P$. The Hilbert operator $\varepsilon x.\ P$ returns an arbitrary x that makes $P\ x$ true. Further, there is the logical equality $\_ = \_$ of type $[\alpha, \alpha] \to$ bool.

This core language can be extended to large libraries comprising Cartesian product types $\_ \times \_$ with the usual projections fst and snd as well as type sums $\_ + \_$, with the injections Inl and Inr. The set type α set can be introduced isomorphic to the function space $\alpha \Rightarrow$ bool, i. e., to characteristic functions, and a typed set theory is introduced with the usual operators, e. g., $\_ \in \_$, $\_ \cup \_$, $\_ \cap \_$.

The HOL type constructor $\tau_\perp$ assigns to each type τ a type *lifted* by $\perp$. The function $\llcorner\_\lrcorner : \alpha \Rightarrow \alpha_\perp$ denotes the injection, the function $\ulcorner\_\urcorner : \alpha_\perp \Rightarrow \alpha$ its inverse for defined values. Partial functions $\alpha \rightharpoonup \beta$ are just functions $\alpha \Rightarrow \beta_\perp$ over which the usual concepts of domain dom f and range ran f are defined. Moreover, on each type $\alpha_\perp$ a test for definedness is available via $\text{def}\, x \equiv (x \neq \perp)$.

3 A Formal Semantics of OCL in Isabelle/HOL

In this section, we formalize a core subset of UML and OCL. While this presentation of the UML/OCL semantics is largely equivalent to the formal semantics presented in the official OCL standard [24, Appendix A] (which is based on [28]), we use a level of abstraction here that is in between the "paper-and-pencil" semantics presented in [24,29] and our machine-checked version [9,6]. In particular, our presentation differs in the following details:

- while [24, Appendix A] uses several interpretation functions $I[\![\_]\!]$ mapping *syntactic expressions e* of the language and *contexts* τ to values in some semantic domain $\mathscr{D}$ and functions over them, we avoid $I[\![\_]\!]$ and state these values and functions directly. Technically, we use an implementation technique called *shallow embedding* [5].
- Instead of one untyped semantic domain, we use a type-indexed family $\mathscr{D}_\tau$ and thus use a type discipline in our meta-language. These types τ are formal representations of OCL types handled in [24, Appendix A] only partially.

Moreover, our formalization follows the formal semantics presented in the non-normative part of the OCL standard [24, Appendix A] (which, in itself, is based on [29]). The (minor) differences of HOL-OCL to the normative part of the OCL standard [24] are discussed in [6].

3.1 Validity and Evaluations

The topmost goal of the formal semantics is to define the *validity statement*:

$$(\sigma, \sigma') \models P, \tag{3}$$

where σ is the pre-state and σ' the post-state of the underlying system and P is a Boolean expression (a *formula*). The assertion language of P is composed of

1. operators on built-in data structures such as Boolean or set,
2. operators of the user-defined data-model such as accessors, type-casts and tests, and
3. user-defined, side-effect-free methods.

Informally, a formula P is valid if and only if its evaluation in the context (σ, σ') yields true. As all types in HOL-OCL are extended by the special element $\bot$ denoting undefinedness, we define formally:

$$(\sigma, \sigma') \models P \equiv \left(P(\sigma, \sigma') = \lfloor \mathrm{true} \rfloor\right). \tag{4}$$

Since all operators of the assertion language depend on the context (σ, σ') and result in values that can be $\bot$, all expressions can be viewed as *evaluations* from (σ, σ') to a type $\tau_\bot$. Consequently, all types of expressions have a form captured by the following type abbreviation

$$V(\tau) := \mathrm{state} \times \mathrm{state} \to \tau_\bot, \tag{5}$$

where state stands for the system state and state $\times$ state describes the pair of pre-state and post-state and _ := _ denotes the type abbreviation.

3.2 Strict Operations

Following common terminology, an operation that returns $\bot$ if one of its arguments is $\bot$ is called *strict*. The majority of the operations is strict, e. g., the Boolean negation is formally

presented as:

$$I[\![\neg X]\!]\tau = \begin{cases} \lfloor \ulcorner \neg I[\![X]\!]\tau \urcorner \rfloor & \text{if } [\![X]\!]\tau \neq \bot, \\ \bot & \text{otherwise}. \end{cases} \tag{6}$$

where $\tau = (\sigma, \sigma')$ and $I[\![\_]\!]$ is just a notation marking the HOL-OCL constructs to be defined. This notation motivated by the definitions in the OCL standard [25]. In our case, $I[\![\_]\!]$ is just the identity, i. e., $I[\![X]\!] \equiv X$. Moreover, we use syntactic overloading: the $\neg\_$ operator on the right has type bool $\to$ bool and refers to the logical negation of HOL, while the $\neg\,\_$ operator on the left has type $V(\text{bool}) \to V(\text{bool})$ and refers to the HOL-OCL negation. The types are different such that confusion is systematically avoided; however, we will use a special highlighting to improve the readability in this presentation. All these operators can be viewed as *transformers on evaluations*.

The binary case of the integer addition is analogous:

$$I[\![X + Y]\!]\,\tau = \begin{cases} \lfloor \ulcorner I[\![X]\!]\,\tau \urcorner + \ulcorner I[\![Y]\!]\,\tau \urcorner \rfloor & \text{if } I[\![X]\!]\,\tau \neq \bot \text{ and } I[\![Y]\!]\,\tau \neq \bot, \\ \bot & \text{otherwise}. \end{cases} \tag{7}$$

Here, the operator $\_+\_$ on the right refers to the integer HOL operation with type $[\text{int}, \text{int}] \to$ int. The type of the corresponding strict HOL-OCL operator $\_ + \_$ is $[V(\text{int}), V(\text{int})] \to V(\text{int})$.

A slight variation of this definition scheme is used for the operators on collection types such as HOL-OCL sets, sequences or bags:

$$I[\![X \cup Y]\!]\tau = \begin{cases} S\lfloor \ulcorner I[\![X]\!]\tau \urcorner \cup \ulcorner I[\![Y]\!]\tau \urcorner \rfloor & \text{if } I[\![X]\!]\tau \neq \bot \text{ and } I[\![Y]\!]\tau \neq \bot, \\ \bot & \text{otherwise}. \end{cases} \tag{8}$$

Here, S ("smash") is a function that maps a lifted set $\lfloor X \rfloor$ to $\bot$ if and only if $\bot \in X$ and to the identity otherwise. Smashedness of collection types is the natural extension of the strictness principle for data structures.

Intuitively, the type expression $V(\tau)$ is a representation of the type that corresponds to the HOL-OCL type τ. Thus, we introduce the following type abbreviations:

$$\texttt{Boolean} := V(\text{bool}), \qquad \texttt{Set}(\alpha) := V(\alpha\,\text{set}), \tag{9}$$

$$\texttt{Integer} := V(\text{int}), \text{ and} \qquad \texttt{Sequence}(\alpha) := V(\alpha\,\text{list}). \tag{10}$$

These abbreviations exemplify the fact that the mapping of an expression E of HOL-OCL type T to a HOL expression E of HOL type T is injective and preserves well-typedness.

3.3 Boolean Operators

There is a small number of explicitly stated exceptions from the general rule that HOL-OCL operators are strict: the strong equality, the definedness operator and the logical connectives.

As a prerequisite, we define the logical constants for truth, absurdity and undefinedness. In the sequel, however, we omit the semantic bracket notation $I[\![\_]\!]$ since it is redundant in our setting. Thus, instead of $I[\![\texttt{T}]\!]\tau = \lfloor \text{true} \rfloor$, we write these definitions as follows:

$$\texttt{T} \equiv \lambda\,\tau.\, \lfloor \text{true} \rfloor, \qquad \texttt{F} \equiv \lambda\,\tau.\, \lfloor \text{false} \rfloor, \text{ and} \qquad \bot \equiv \lambda\,\tau.\, \bot. \tag{11}$$

HOL-OCL has a *strict equality* $\_ \doteq \_$ which is defined similarly to the integer addition. However, for specification purposes, we introduce also a *strong equality* $\_ \mathrel{\triangleq} \_$ which is defined as follows:

$$X \mathrel{\triangleq} Y \equiv \lambda\, \tau.\ (X\, \tau = Y\, \tau)\,, \tag{12}$$

where the $\_ = \_$ operator on the right denotes the logical equality of HOL. The undefinedness test is defined by $\partial\, X \equiv (X \mathrel{\triangleq} \bot)$. The strong equality can be used to state reduction rules like: $\tau \vDash (\bot \doteq X) \mathrel{\triangleq} \bot$.

The OCL standard requires a Strong Kleene Logic. In particular, it defines:

$$X \wedge Y \equiv \lambda\, \tau.\ \begin{cases} \lfloor \ulcorner X\, \tau \urcorner \wedge \ulcorner Y\, \tau \urcorner \rfloor & \text{if } X\, \tau \neq \bot \text{ and } Y\, \tau \neq \bot, \\ \lfloor \text{false} \rfloor & \text{if } X\, \tau = \lfloor \text{false} \rfloor \text{ or } Y\, \tau = \lfloor \text{false} \rfloor, \\ \bot & \text{otherwise}\,. \end{cases} \tag{13}$$

of type $[\texttt{Boolean}, \texttt{Boolean}] \to \texttt{Boolean}$. The other Boolean connectives are defined as follows:

$$X \vee Y \equiv \neg(\neg X \wedge \neg Y) \qquad\qquad X \longrightarrow Y \equiv \neg X \vee Y \tag{14}$$

The logical quantifiers are viewed as special operations on the collection types $\texttt{Set}(\alpha)$ or $\texttt{Sequence}(\alpha)$. Their definition in the OCL standard is very operational and restricted to the finite case; instead, we define the universal quantification as generalization of the conjunction:

$$\big(\forall\, x \in X\,.\ P(X)\big) \equiv \lambda\, \tau.\ \begin{cases} \bot & \text{if } X\, \tau = \bot, \\ \lfloor \forall x \in \ulcorner X\, \tau \urcorner.\ \ulcorner P(\lambda\, \tau.\ x) \tau \urcorner \rfloor & \text{if } \forall x \in \ulcorner X\, \tau \urcorner.\ P(\lambda\, \tau.\ x)\, \tau \neq \bot, \\ \lfloor \text{false} \rfloor & \text{if } \exists x \in \ulcorner X\, \tau \urcorner.\ P(\lambda\, \tau.\ x) \tau = \lfloor \text{false} \rfloor, \\ \bot & \text{otherwise;} \end{cases} \tag{15}$$

and the existential quantification is defined as follows:

$$\big(\exists\, x \in X\,.\ P(x)\big) \equiv \big(\neg\, \forall\, x \in X\,.\ \neg P(x)\big)\,. \tag{16}$$

3.4 An Axiomatization of Object-oriented Data Structures

In the previous sections, we described various built-in operations on datatypes and the logic. Now we turn to several families of operations that the user implicitly defines when stating a class model, e. g., described as a UML class diagram, as logical context of a specification. This is the part of the language where object-oriented features such as type casts, accessor functions, and tests for dynamic types play a role, and this is the issue that makes HOL-OCL remarkably different from specification formalisms such as VDM or Z.

3.4.1 Class Models: The Syntax

In the following, we define the notion of a *class model* precisely. A class model is a tuple $CM = (C, A, M, \_ < \_)$ where:

- C, a set of *class names* containing at least the name of the common superclass, i. e., `OclAny`,
- A, a set of *class attributes*,
- a finite partial map M that assigns to each class identifier a finite partial map assigning attributes to types: $M \equiv C \rightharpoonup (A \rightharpoonup T)$,
- a partial irreflexive order $\_ < \_$ on class names called *class hierarchy*; in particular, we assume $X < \texttt{OclAny}$ for all $X \in \operatorname{dom} M$.

For simplicity, we assume that attribute names are unique throughout this paper; this means that the domains of two elements of the range of M are always pairwise disjoint. For $c_i < c_j$, we will say that the class c_i is a *subclass* of class c_j; the restriction that any class is a subclass of `OclAny` can be imposed without loss of generality. Furthermore, we assume a set of types inductively defined as follows:

$$\begin{aligned} T := C \mid \texttt{OclAny} \mid \texttt{Boolean} \mid \texttt{Integer} \mid \texttt{Real} \mid \texttt{String} \\ \mid \texttt{Sequence}(T) \mid \texttt{Set}(T) \mid \texttt{Bag}(T) \mid T \mathrel{-\!\!>} T\,. \end{aligned} \tag{17}$$

All classes, referenced by their names, induce an own type, the *class type*, that represents the type of all *objects* of this class. Objects are typed pieces of data that contain values for each attribute of the associated class.

3.4.2 Class Models: The Induced Signature

A class model induces several families of functions, which we will axiomatize in the following. This representation is simplified compared to the technique used in HOL-OCL. HOL-OCL constructs a model for this axiom system for a given class system by compiling it into a sequence of conservative definitions. The presented "axioms" are then derived from them. Due to space limitations, we will not go into detail and refer the interested reader to [6, 10] for details.

The families of functions induced by a class model CM comprise:

1. for each $C \in \operatorname{dom} M$, and each attribute $a \in \operatorname{dom}(M\,C)$ of type T, there is an attribute accessor function $\_.a$ as well as an attribute accessor function $\_.a\,\texttt{@pre}$ of type $C \mathrel{-\!\!>} T$,
2. a $C \in \operatorname{dom} M$ indexed family of overloaded *type-casts*: $\_{[C]}$ of type $X \mathrel{-\!\!>} C$ for all $X < C$ or $C < X$,
3. a $C \in \operatorname{dom} M$ indexed family of tests for the *dynamic type* of an object: $\mathrm{isType}_C\,\_$, and
4. a $C \in \operatorname{dom} M$ indexed family of tests for the *dynamic kind* of an object: $\mathrm{isKind}_C\,\_$.

Ad item 1: Accessor functions always return typed *objects*, or values, and never "references"; however, references are used *internally* in the semantic model of the object construction. The accessor functions return the value of the attribute of a given object if the attribute has type, e. g., `Boolean`, `Integer`, `Real`. If the attribute type has class type, the attribute contains a reference which is referenced in the state or pre-state, yielding again an object. Following the injectivity principle of the representation map, we give an accessor $\_.a$ (or: $\_.a\,\texttt{@pre}$, referencing in the pre-state) of HOL-OCL type $C \mathrel{-\!\!>} T$ the HOL type of an evaluation transformer: $V(C) \to V(T)$.

Ad item 2: Type-casts are the omnipresent glue in expressions of object-oriented languages. They are used to *implement* subtyping by interfacing objects up and down the class

hierarchy to fit them in as argument of an operation. Both up and down casts have to be semantically lossless; for example, it must be possible to cast an arbitrary object up to `OclAny`, include it into a set of `Set(OclAny)`, exclude it later and cast again to the identical object (this is the way generic datatypes are implemented in, e. g., Java).

Ad item 3: Applying type-casts modifies the static type of an object, i. e., the type inferred by static type inference. To enable down-casts appropriately, however, it must be possible to reconstruct for an object which type it had "originally," i. e., at creation time. The test $\mathrm{isType}_C\, obj$ holds if and only if the *dynamic type* of *obj* is *C*.

Ad item 4: A relaxation of the dynamic type test is the *dynamic kind* test which holds if and only if the dynamic type of *obj* is *C* or a subtype of *C*.

3.4.3 Class Models: An Axiomatization

First of all, all functions of the induced signature are strict. This means that the following scheme of rules hold:

$$\bot.a = \bot \qquad \bot_{[C]} = \bot \qquad \mathrm{isType}_C\, \bot = \bot \qquad \mathrm{isKind}_C\, \bot = \bot \tag{18}$$

for all attributes *a* of a class model *CM* and all $C \in \mathrm{dom}\, M$. These equalities are HOL equalities and both left and right hand sides are evaluations, i. e., functions depending on the context. By extensionality, these equalities express that both sides are congruent for all contexts (see also discussion in Section 4).

Furthermore, the following rule schemes express that the dynamic type remains unchanged while casting:

$$\mathrm{isType}_C\, obj_{[C']} = \mathrm{isType}_C\, obj \qquad \mathrm{isKind}_C\, obj_{[C']} = \mathrm{isKind}_C\, obj \tag{19}$$

for all $C, C' \in \mathrm{dom}\, M$.

Moreover, we can "re-cast" an object safely, i. e., up and down casts are idempotent. However, casting an object deeper in the subclass hierarchy than its dynamic type results in undefinedness. Furthermore, casting is transitive:

$$\frac{\tau \models \mathrm{isType}_B\, obj}{\tau \models ((obj_{[A]})_{[B]}) \triangleq obj} \qquad \frac{\tau \models \mathrm{isType}_A\, obj}{\tau \models obj_{[B]} \triangleq \bot} \tag{20}$$

$$\frac{\tau \models \mathrm{isType}_C\, obj}{\tau \models (obj_{[B]})_{[A]} \triangleq obj_{[A]}} \qquad \frac{\tau \models \mathrm{isType}_A\, obj}{\tau \models \mathrm{isKind}_B\, obj} \tag{21}$$

where we assume $A, B, C \in \mathrm{dom}\, M$ and $C < B < A$.

The reason why these rules look slightly more complicated than, e. g., the strictness rules above, consists in the fact that these are conditional rules where all parts must refer to the same context τ.

Accessor functions to attributes that are not defined in a class are syntactically illegal and ruled out by the typing discipline.

3.5 Class Invariants

A *class invariant* of class C is a formula $inv_C(obj)$ that is satisfied by all objects in a state, i. e., a partial map of object-identifiers to objects of type oid $\rightharpoonup$ `OclAny`. The traditional way to define invariants is via an own, class-indexed family of operators:

$$C\texttt{::allInstances()}(\sigma,\sigma') \equiv \{obj_{[C]} \in \operatorname{ran}\sigma' \mid \operatorname{isType}_C\, obj\} \tag{22}$$

and a conjunction of the formulae:

$$\forall x \in (C\texttt{::allInstances()}).\ \operatorname{inv}_C(x) \tag{23}$$

which constrains the set of possible states $\sigma :: state$ to the *valid states*.

We use a special linked list as a simple example: we assume a class `Node` with an Integer attribute `i` and one attribute `next` of type `Node`. We want to characterize the subset of `Node` objects within a state where in each node the value of `i` is greater than `5` and where each value of `next` is defined and again in this set. This is expressed by a subclass `SNode` with the following class invariant:

```
context SNode
  inv defined: self.i > 5 ∧ isKind_SNode self.next
```

As a result, $\operatorname{inv}_{\text{SNode}}$ enjoys the recursive equation:

$$\operatorname{inv}_{\text{SNode}}(obj :: \text{SNode}) = obj.\text{i} > 5 \wedge \partial\, obj.\text{next} \wedge \operatorname{inv}_{\text{SNode}}(obj.\text{next}) \tag{24}$$

The definedness of the `next` attribute is a consequence of the fact that the invariant must be valid, implying that $\operatorname{isKind}_{\text{SNode}}\, self.\text{next}$ must be valid, implying that $self.\text{next}$ must be defined. We believe that this implicitness with respect to definedness is a good motivation for using a three-valued logic. HOL-OCL offers specialized support for the derivation of these equations which are crucial for many derivations (see [9] for details).

3.6 Operation Specifications

The semantics of an operation specification is a big-step transition-system semantics (similar to Z) and *not* a small-step Hoare-style semantics; it is therefore constructed by the *conjunction* of the validity of preconditions and postcondition and not by their implication. This means that an operation specification:

```
context C :: op(p_1 : T_1, ..., p_n : T_n) : T_{n+1}
  pre:  P
  post: Q
```

is presented by a straightforward conversion into the definition:

$$\operatorname{op}_C\, self\ p_1 \cdots p_n \equiv \lambda\, \tau.\ \varepsilon\, result.\ \tau \models P\, self\ p_1 \cdots p_n \wedge \tau \models Q\, self\ p_1 \cdots p_n\, result \tag{25}$$

This conversion makes the implicit input parameter *self* and output parameter *result* explicit.

Moreover, we allow the overriding of operations. This means that for each class $C_m < C_{m-1}, \ldots, C_2 < C_1$ in a class model, a new operation specification can be given that is defined

on the corresponding subtypes of *self*. For a given class model, the combined specification operation for a method invocation is defined as:

$$\begin{aligned}
\mathrm{op}\ \mathit{self}\ p_1 \ldots p_n \equiv \quad & \texttt{if}\ \mathrm{isKind}_{C_m}\ \mathit{self}\ \texttt{then}\ \mathrm{op}_{C_m}\ \mathit{self}\ p_1 \ldots p_n \\
& \texttt{else if}\ \mathrm{isKind}_{C_{m\text{-}1}}\ \mathit{self}\ \texttt{then}\ \mathrm{op}_{C_{m-1}}\ \mathit{self}\ p_1 \ldots p_n \\
& \qquad\qquad \vdots \\
& \texttt{else if}\ \mathrm{isKind}_{C_1}\ \mathit{self}\ \texttt{then}\ \mathrm{op}_{C_1}\ \mathit{self}\ p_1 \ldots p_n \\
& \texttt{endif} \cdots \texttt{endif}\ \texttt{endif}
\end{aligned}$$

This order of resolving the invocation overloaded operations implements *late-binding*, a resolution mechanism widely used in object-oriented programming languages such as Java or C#.

3.7 Discussion: Simplifications Underlying this Presentation

The construction presented in this section is simplified in several aspects compared to the construction used in the HOL-OCL [9] system. In particular:

1. The presented construction is based on a specific closed world assumption: it is implicitly assumed that a class model consists only of its fixed number of classes (and thus also on attributes, methods, ...). This is a consequence of the simplistic construction presented here: i. e., for each given class model, the corresponding set of axioms is generated. If the class model is extended or modified, another set of axioms will be generated. In practice, this means that all proofs built upon a class model must be revalidated after an extension, which limits the usefulness of the technique. In HOL-OCL, a more refined technique is therefore used which constructs a model entirely based on conservative definitions which is in fact extensible, i. e., amenable to a form of modular verification. Since the details of the construction are quite involved, we constrained ourselves to this axiomatic fragment.
2. Moreover, the HOL-OCL class model compiler also generates functions that allow to construct a state; i. e., there is an infrastructure to create objects, update attributes in them, and determine even their *object-identifier* or reference within a state.

These more powerful constructions are described elsewhere [6, 10].

4 Proof Calculi for OCL

In this section, we present several deduction systems for HOL-OCL. In particular, we define two equational calculi well-suited for interactive proofs and a tableaux calculus geared towards automatic reasoning. All rules we present are derived within Isabelle from the semantic definitions introduced in Section 3. Therefore, we can guarantee the logical soundness of all these rules with respect to the core logic. These three calculi were used to instantiate Isabelle's (two-valued) generic proof-procedures yielding in decision procedures for fragments of OCL.

4.1 Validity and Universal Congruence

Since OCL is a three-valued logic, each OCL expression either evaluates to true (`T`), false (`F`), or undefined ($\bot$). Thus, the following theorem holds:

$$(\tau \models A) \vee (\tau \models \neg A) \vee (\tau \models \neg(\delta\, A)). \tag{26}$$

The natural question arises, under which conditions two formulae are equivalent for *all* contexts τ, e. g., $\forall\tau.\, A\,\tau = B\,\tau$. Since the logical equality of HOL enjoys extensionality and since all OCL formulae are evaluations, we can express this simply by:

$$A = B. \tag{27}$$

Equations on OCL formulae are called *universal congruences*; so far, we have seen equalities of this form in Equation 18, in Equation 19 and all reduction rules for strict functions like $f\ \bot = \bot$. Due to Equation 26, we can establish the following link between validity and universal congruence:

$$\frac{\bigwedge\tau.\,(\tau \models X) = (\tau \models Y) \qquad \bigwedge\tau.\,\big(\tau \models \neg(\delta\, X)\big) = \big(\tau \models \neg(\delta\, Y)\big)}{X = Y}\,. \tag{28}$$

This rule is sound also the other way round (for trivial reasons). Note, furthermore, that the underlying choice for "evaluation to truth and evaluation to undefined" is arbitrary; whenever evaluations of two formulae agree on two out of the three cases, they are universally congruent.

4.2 The Universal Equational Calculus

The basis of Universal Equational Calculus (UEC), see Table 1, are Horn-clauses over universal congruences which can be applied in arbitrary OCL expressions. A *proof* of a formula ϕ in UEC is simply its reduction to `T`, since the following theorem holds:

$$(\tau \models \phi) = (\tau \models \mathtt{T}) = (\lfloor\mathrm{true}\rfloor = \lfloor\mathrm{true}\rfloor) = \mathrm{true}. \tag{33}$$

Based on the semantic definitions for the logical operators, it is not difficult to derive the laws of the surprisingly rich algebraic structure of Strong Kleene Logic: both $\_ \wedge \_$ and $\_ \vee \_$ enjoy associativity, commutativity and idempotency. The logical operators also satisfy both distributivity and the de Morgan laws. It is essentially this richness and algebraic simplicity that can be exploited for normal-form computations as well as "proofs-by-hand" such as the example:

$A_1 \wedge \cdots \wedge A_k \wedge \delta\ B \wedge A_{k+1} \wedge \cdots \wedge A_n \longrightarrow \delta\ B$

$= A_1 \wedge \cdots \wedge A_n \longrightarrow (\delta\ B \longrightarrow \delta\ B)$

and since $\delta\,\delta\ B = \mathtt{T}$, we can conclude

$= A_1 \wedge \cdots \wedge A_n \longrightarrow \mathtt{T}$

and thus, reduce our proof goal to:

$= \mathtt{T}$.

$$\mathtt{F} \wedge X = \mathtt{F} \qquad \mathtt{T} \vee X = \mathtt{T}$$
$$\mathtt{T} \wedge X = X \qquad \mathtt{F} \vee X = X$$
$$X \wedge X = X \qquad X \vee X = X$$
$$X \wedge Y = Y \wedge X \qquad X \vee Y = Y \vee X$$
$$X \wedge (Y \wedge Z) = (X \wedge Y) \wedge Z \qquad X \vee (Y \vee Z) = (X \vee Y) \vee Z$$
$$\neg(\neg X) = X \qquad (X \wedge Y) = \neg(\neg X \vee \neg Y)$$
$$(X \vee Y) \wedge Z = (X \wedge Z) \vee (Y \wedge Z) \qquad (X \wedge Y) \vee Z = (X \vee Z) \wedge (Y \vee Z)$$
$$\neg(X \wedge Y) = \neg X \vee \neg Y \qquad \neg(X \vee Y) = \neg X \wedge \neg Y$$

(a) Lattice

$$\partial\,\mathtt{F} = \mathtt{T} \qquad \partial\,\mathtt{T} = \mathtt{T} \qquad \partial\,\bot = \mathtt{F}$$
$$\partial\,\partial X = \mathtt{T} \qquad \partial(\neg X) = \partial X$$
$$\partial(X \wedge \partial X) = \mathtt{T} \qquad \partial(\neg X \wedge \partial X) = \mathtt{T}$$
$$\partial(X \triangleq Y) = \mathtt{T} \qquad \partial(X \doteq Y) = \partial X \wedge \partial Y$$
$$\partial(\mathtt{if}\ \mathrm{X}\ \mathtt{then}\ \mathrm{Y}\ \mathtt{else}\ \mathrm{Z}\ \mathtt{endif}) = \partial \mathrm{X} \wedge (\mathrm{X} \wedge \partial \mathrm{Y} \vee \neg \mathrm{X} \wedge \partial \mathrm{Z})$$
$$\partial(\mathrm{X} \wedge \mathrm{Y}) = (\partial \mathrm{X} \wedge \partial \mathrm{Y}) \vee \neg \mathrm{X} \vee \neg \mathrm{Y}$$
$$\partial(\mathrm{X} \vee \mathrm{Y}) = (\partial \mathrm{X} \wedge \partial \mathrm{Y}) \vee \mathrm{X} \vee \mathrm{Y}$$
$$\partial(\mathrm{X} \longrightarrow \mathrm{Y}) = (\partial \mathrm{X} \wedge \partial \mathrm{Y}) \vee \neg \mathrm{X} \vee \mathrm{Y}$$

(b) Strong definedness rules.

$$\mathtt{if}\ \bot\ \mathtt{then}\ \mathrm{Y}\ \mathtt{else}\ \mathrm{Z}\ \mathtt{endif} = \bot$$
$$f \bot = \bot \qquad f \bot Y = \bot \qquad f X \bot = \bot$$
$$\partial\, f X = \partial X \qquad \partial\, f X Y = \partial X \wedge \partial Y$$

(c) Strictness/Definedness Rules for total strict operations f.

$$\mathtt{if}\ \mathtt{T}\ \mathtt{then}\ \mathrm{Y}\ \mathtt{else}\ \mathrm{Z}\ \mathtt{endif} = Y \qquad \mathtt{if}\ \mathtt{F}\ \mathtt{then}\ \mathrm{Y}\ \mathtt{else}\ \mathrm{Z}\ \mathtt{endif} = Z$$
$$X \longrightarrow \mathtt{F} = \neg X \qquad X \longrightarrow \mathtt{T} = \mathtt{T} \qquad \mathtt{F} \longrightarrow X = \mathtt{T} \qquad \mathtt{T} \longrightarrow X = X$$
$$\mathrm{X} \longrightarrow \mathrm{Y} = \neg \mathrm{X} \vee \mathrm{Y}$$
$$\mathrm{X} \longrightarrow (\mathrm{Y} \wedge \mathrm{Z}) = (\mathrm{X} \longrightarrow \mathrm{Y}) \wedge (\mathrm{X} \longrightarrow \mathrm{Z})$$
$$\mathrm{X} \longrightarrow (\mathrm{Y} \vee \mathrm{Z}) = (\mathrm{X} \longrightarrow \mathrm{Y}) \vee (\mathrm{X} \longrightarrow \mathrm{Z})$$
$$(\mathrm{X} \wedge \mathrm{Y}) \longrightarrow \mathrm{Z} = \mathrm{X} \longrightarrow (\mathrm{Y} \longrightarrow \mathrm{Z})$$
$$(\mathrm{X} \vee \mathrm{Y}) \longrightarrow \mathrm{Z} = (\mathrm{X} \longrightarrow \mathrm{Z}) \wedge (\mathrm{Y} \longrightarrow \mathrm{Z})$$
$$\mathrm{X} \longrightarrow (\mathrm{Y} \longrightarrow \mathrm{Z}) = \mathrm{Y} \longrightarrow (\mathrm{X} \longrightarrow \mathrm{Z})$$
$$\frac{\partial X = \mathtt{T}}{(X \longrightarrow X) = \mathtt{T}}$$

(d) Logic

Table 1 The Universal Equational Calculus.

$$\frac{}{\tau \models a \triangleq a} \qquad \frac{\tau \models a \triangleq b}{\tau \models b \triangleq a} \qquad \frac{\tau \models a \triangleq b \quad \tau \models b \triangleq c}{\tau \models a \triangleq c}$$

$$\frac{\tau \models a \triangleq b \quad \tau \models Pa \quad \mathrm{cp}(P)}{\tau \models Pb}$$

Table 2 Quasi-Equational Theory.

$$\frac{}{\mathrm{cp}(\lambda X.\, X)} \qquad \frac{}{\mathrm{cp}(\lambda X.\, c)}$$

$$\frac{\mathrm{cp}\, P}{\mathrm{cp}(\lambda X.\, f_1(PX))} \qquad \frac{\mathrm{cp}\, P \qquad \mathrm{cp}\, P'}{\mathrm{cp}\big(\lambda X.\, f_2(PX)(P'X)\big)}$$

$$\frac{\mathrm{cp}(P) \qquad \bigwedge x.\ \mathrm{cp}(P'x)}{\mathrm{cp}(\lambda X.\ \forall\, (PX)\, (\lambda x.\, (P'\, xX)))}$$

Where f_1 and f_2 are strict operators that are defined by one of the three definition schemes described in Section 3.2, a logical connective, or a quantifier.

Table 3 The Core Context Passing Rules.

4.3 Reasoning over OCL-Equalities

The UEC introduced in Section 4.2 cannot be complete: some facts will only hold in *some* contexts τ, not in all of them. Moreover, expressing foundational facts on the crucial strong equality $\_ \triangleq \_$ is not possible within UEC, e. g., as equality on evaluations. Due to the handling of contexts, the usual congruence properties of an equality can only be approximated (see Table 2). This is backed up by the well-formedness predicate $\mathrm{cp}(P)$ (called *P is context-passing*) which requires that the context τ is unchanged "on its way through P." Formally, we can define context passing $\mathrm{cp}(P)$:

$$\mathrm{cp}(P) \equiv \exists E.\, \forall X\, \tau.\, PX\, \tau = E(X\, \tau)\,, \tag{34}$$

i. e., each term P (a transformer on evaluations) containing a sub-term X (an evaluation) must be replaceable by some E which just takes the value of X constructed by evaluating it by τ.

At first sight, the handling of $\mathrm{cp}\_$ seems to be infeasible; however, it can be established by a simple subcalculus that decides this property for all expressions P which are only λ-abstractions of terms built *uniquely* by OCL operators (see Table 3).

The importance of the strong equality becomes apparent with the following rules, which establish a link to strict equality, validity, invalidity and undefinedness:

$$\frac{\tau \models a \doteq b}{\tau \models a \triangleq b}\,, \tag{35}$$

$$\tau \models A = (\tau \models A \triangleq \mathtt{T})\,, \tag{36}$$

$$\tau \models \neg A = (\tau \models A \triangleq \mathtt{F})\text{, and} \tag{37}$$

$$(\neg\,\tau \models \partial\, A) = \tau \models \neg(\partial\, A) = (\tau \models A \triangleq \bot)\,. \tag{38}$$

The trichotomy rule Equation 26, the rules Equation 36, Equation 37, and Equation 38, in connection with the substitutivity (see Table 2) allow for case-splits in arbitrary subformulae.

4.4 A Tableaux Calculus on Judgements

The tableaux methodology is a popular approach to design and implement proof-procedures. Originally, tableaux methods were geared towards first-order theorem proving, in particular for non-clausal formulae accommodating equality. Nevertheless, renewed research activity is being devoted to investigating tableaux systems for intuitionistic, modal, temporal and many-valued logics, as well as for new families of logics, such as non-monotonic and substructural logics. Many of these recent approaches are based on a special labeling technique on the level of judgments, called labeled deduction [12,32]. Of course, labeling can also be embedded into a higher-order, classical meta-logic. For example, the conjunction introduction and elimination rules can be presented in natural style supported by Isabelle:

$$\frac{\tau \models A \qquad \tau \models B}{\tau \models (A \wedge B)} \quad \text{or} \quad \frac{\tau \models A \wedge B \qquad \begin{array}{c}[\tau \models A, \tau \models B]\\ \vdots \\ R\end{array}}{R}\,. \tag{39}$$

The operational effect of these rules in backward-style derivations is to split a validity judgement into two or to replace a judgement with a conjunction in the assumption list by the two simpler judgements. In effect, formulae are transformed into clauses consisting of atomic validity judgments containing no further logical connectives.

Tableaux calculi for strong Kleene Logic based on labeled deduction have been extensively studied [17,15,14]. We also derived a tableaux calculus for OCL [9], but the approach turned out inefficient even for small formulae.

The problem becomes apparent when considering the rules in Table 1b on page 14: establishing the definedness, which is an omnipresent side-condition to many rules, leads to many redundant case-splits which again result in definedness-reasoning; by using clever rules and forward inference techniques, this effect can only partly be compensated. Furthermore, the process of finding unifiers for quantifiers by a multitude of closing rules turned out to be difficult to implement inside Isabelle.

4.5 A Conversion Calculus for OCL

Table 4 shows the conversion calculus which we derived as a practical automated reasoning procedure. This conversion calculus allows to convert the three-valued reasoning into classical reasoning in HOL; the resulting expressions can then be handled by the standard tableaux procedures of Isabelle. Thus, if we can establish effectively once and for all if all related sub-expressions are defined, we can convert a formula into a classical logical expression containing only two-valued judgements.

$$\frac{\tau \models \partial A}{(\tau \models \neg A) = (\neg \tau \models A)} \qquad \frac{\tau \models \partial A \quad \tau \models \partial B}{(\tau \models A \wedge B) = (\tau \models A \wedge \tau \models B)}$$

$$\frac{\tau \models \partial A \quad \tau \models \partial B}{(\tau \models A \vee B) = (\tau \models A \vee \tau \models B)} \qquad \frac{\tau \models \partial A \quad \tau \models \partial B}{(\tau \models A \longrightarrow B) = (\tau \models A \longrightarrow \tau \models B)}$$

$$\frac{\tau \models \partial (S :: \mathtt{Set}(\beta)) \quad \mathrm{cp}\, P}{(\tau \models \forall x \in S \,.\, P x) = (\forall x.\ \tau \models x \in S \longrightarrow \tau \models P x)}$$

$$\frac{\tau \models \partial (S :: \mathtt{Set}(\beta)) \quad \mathrm{cp}\, P}{(\tau \models \exists x \in S \,.\, P x) = (\exists x.\ \tau \models x \in S \wedge \tau \models P x)}$$

Table 4 The Translation Rules.

4.6 A Note on Quantifiers

In the following, we discuss the extension of the propositional fragment by bounded quantifiers introduced for collections. For brevity, we will concentrate on the quantifiers on sets, i. e., $\mathtt{Set}(\tau)$.

First, we present some universal equalities of the universal quantifiers, which also satisfy the usual context passing rules in Table 3. With respect to strictness rules, the definitions of the quantifiers follow the usual scheme:

$$\big(\forall x \in \bot \,.\, P x\big) = \bot, \tag{41}$$

$$\big(\exists x \in \bot \,.\, P x\big) = \bot, \tag{42}$$

$$\big(\forall x \in \emptyset \,.\, P x\big) = \mathtt{T}, \text{ and} \tag{43}$$

$$\big(\exists x \in \emptyset \,.\, P x\big) = \mathtt{F}. \tag{44}$$

Second, for the defined cases, we have again conversion rules that allow for a semantic mapping of the HOL-OCL quantifiers to their classical HOL counterparts (see Table 4).

Moreover, if $x \in S$ is valid, we know that x must be defined. This is a characteristic property of smashed sets that yields the following property:

$$\frac{\tau \models x \in S}{\tau \models \partial x}\,. \tag{45}$$

Thus, for smashed sets, bound variables are always defined.

4.7 Automated Deduction.

The question arises, how can these calculi be combined to proof procedures that decide certain fragments of the language, i. e., to what extent can automated proof support be provided for HOL-OCL? This question is of vital importance for the practicability of a proof environment. We outline two procedures implemented in HOL-OCL here, one for the predicative fragment, one for equational reasoning.

4.7.1 A Practical Decision Procedure for the Predicative Fragment

As briefly discussed in Section 4.4, a direct implementation as a specialized tableaux calculus is difficult, and the theoretic results are discouraging. However, deciding that an expression is *defined* is in practice surprisingly well-behaved. This is a consequence of the fact that the vast majority of operations in HOL-OCL are total and strict.

Moreover, in practical relevant situations (such as proving that some ϕ follows from the class invariants), $\_ \wedge \_$ is the predominant logical connective; this fact can be exploited by normal form computations using Equation 39.

The fact that definedness is often well-behaved, raises the question what balance between forward and backward reasoning avoids most redundancy. As forward-inference component, we decided to case-split over the definedness of all *free variables* occurring in the formula. In principle, this results in exponentially many sub-formulae (called *splinters*); due to Equation 38, the substitutivity rule (see Table 2), and the collection of strictness rules of Table 1c, they can be drastically simplified. In particular, this simplified formulae contain only variables x for which the definedness is explicit: $\tau \models \partial\, x$. Over the normalized splinters, we apply the conversion rules Table 4; the decision of definedness of terms is now just linear over the size of terms. Thus, we achieved a (usually small) number of converted splinters which are ordinary classical first-order formulae to be handled by Isabelle's standard two-valued tableaux procedures.

The rules for cp are pre-computed for each HOL-OCL operator, such that deciding this side-condition is linear in worst-case and logarithmic in the average case in the size of the term. It only involves trivial matches.

4.7.2 Equational Reasoning for Strong Logical Equality

The universal congruence rules can be directly processed by the standard rewriting mechanism of Isabelle; the question is how to handle $\_ \triangleq \_$ effectively. Besides the obvious approach (applying substitutivity in a specialized procedure, and repeating this for each rewrite rule), we developed a more efficient technique. The key observation is that property of being context-passing can be made explicit by the following rule:

$$\big(\tau \models P\,X\big) = \big(\tau \models (K(P(K(X\,\tau))\tau))\big)\,, \tag{46}$$

where K is the K-combinator $(\lambda\, x\, y.\ x)$ and where we require $\mathrm{cp}(P)$. Applying this rule exhaustively to an evaluation ϕ leads to the *cp-normal-form*, and obviously, the transformation is reversible. Interestingly, rules like Equation 20 can be converted into:

$$\frac{\mathrm{isType}_{\mathrm{B}}\,\big(K(obj\,\tau)\big)\,\tau = \mathtt{T}\,\tau}{\Big(\mathrm{K}\Big((\mathrm{K}(obj\,\tau))_{[\mathrm{A}]}\ \tau\Big)_{[\mathrm{B}]}\Big)\,\tau = obj\,\tau}\,, \tag{47}$$

i. e., into a conventional conditional equality. In our approach, rewrite rules as well as proof-states are transformed into cp normal-form, then rewritten by standard simplification, and converted back. Both transformations are linear in the size of the terms; the matching is slowed down since the size of matches doubles. However, as a whole, these costs are neglectable and the resulting simplification procedure is similarly powerful as the original Isabelle simplifier.

5 Applications

In the previous sections, we described a formal, machine-checked semantics for OCL and derived calculi suited for automated reasoning. In this section, we apply this theory: we present concepts and implementation of a methodology on top of HOL-OCL, enforcing a particular use of our language. At present, the following aspects of the methodology are supported:

1. an *analysis method* enforcing a certain form of well-formedness considered pragmatically useful, and
2. a formal *refinement notion* allowing to convert abstract class models into more concrete ones.

Since our refinement notion is transitive, an original abstract design can be converted stepwise into a version that can be converted to code automatically. We also present an implementation of these concepts in the Isabelle framework, allowing to insert "analytical commands" in a proof-document resulting in proof-obligations to be discharged in the sequel; the technology is further described in [33].

5.1 Proving Consistency

When capturing the requirements for a larger software system, the problem arises how to detect potential inconsistencies, contradictions or redundancies in larger numbers of class invariants or method specifications. Thus, prior to any implementation or refinement attempt, there is the need for a *consistency analysis* of the specification.

In the following, we concentrate on a specific kind of consistency that is required by the data refinement methodology we present later. Our refinement methodology has both syntactic (also called *well-formedness* requirements) and semantic requirements.

On the syntactical side, we require for public operations that the return value and their arguments are either basic datatypes (e. g., `Integer`, `String`) or public classes. We call a class *public*, if it contains a least one public attribute or operation; classes that do neither contain attributes nor operations are public by default.

On the semantic side, we need constraints on states σ, not state transitions τ, for the first time. Instead of using syntactic side-conditions (e. g., as in [24, Appendix A]) like "the assertion does not contain the `@pre`-operator," we use a slightly more general semantic characterization which is amenable in calculi. As a prerequisite, we define two assertions ϕ, ϕ' *pre-state equivalent* in σ, written $(\sigma \models \phi) \overset{\text{pre}}{=} (\sigma \models \phi')$:

$$(\sigma \models \phi) \overset{\text{pre}}{=} (\sigma \models \phi') \iff \forall x, y.\ \big((\sigma, x) \models \phi\big) = \big((\sigma, y) \models \phi'\big), \tag{48}$$

i. e., all post-states x and y are irrelevant. For example, this is the case for assertions ϕ,ϕ' where all accessors occurring in them are accessing the pre-state (i. e., using `_@pre`) and where $\phi = \phi'$. Analogously, we define the concept of *post-state equivalence*, written $(\sigma \models \phi) \overset{\text{post}}{=} (\sigma \models \phi')$. Moreover, we introduce the notion *pre-state validity*:

$$(\sigma \models_{\text{pre}} \phi) \iff (\sigma \models \phi) \overset{\text{pre}}{=} (\sigma \models \texttt{T}) \tag{49}$$

and also analogously *post-date validity*: $(\sigma \models_{\text{post}} \phi)$.

Finally, we define a syntactic transformation $\_{\text{pre}}$ of assertions to support certain syntactical conventions of OCL; ϕ_{pre} results from ϕ by substituting all accessor functions by

their _`@pre`-counterparts. For `@pre`-free assertions (e. g., preconditions and invariants), the following property can be proven automatically:

$$(\sigma \models_{\text{post}} \phi) = (\sigma \models_{\text{pre}} \phi_{\text{pre}}). \tag{50}$$

Recall that a state σ is a partial map from object-identifiers to objects; following the OCL standard, we call states *valid* if and only if each object in its range satisfies the class invariants. We write V for the set of valid states. The empty state λ oid. $\bot$ is always in V.

Now we can describe the proof-obligations of a consistent class model conceptually. For a consistent UML/OCL package we require:

1. there must exist a state satisfying the class invariants that contains an object for each public class, i. e.,

$$\exists \sigma \in V, a_1, \ldots, a_n.\ \sigma \models_{\text{post}} \text{inv}_{C_1}(a_1) \wedge \ldots \wedge \sigma \models_{\text{post}} \text{inv}_{C_n}(a_n)$$

   where $C_1, \ldots, C_n$ are the public classes of the class model and $\text{inv}_{C_1}, \ldots, \text{inv}_{C_n}$ are the corresponding class invariants.
2. For all operations, there must be a pre-state satisfying the class invariants and input variables satisfying the precondition, i. e.,

$$\exists \sigma \in V, p_1, \ldots, p_n.\ \sigma \models_{\text{pre}} (\text{pre}_{op}\ p_1 \ldots p_n)_{\text{pre}}$$

   for all public operations op with arguments $p_1, \ldots, p_n$ of the class model.
3. For all operations, for each a pre-state satisfying the class invariants and all input variables satisfying the precondition, there must be a result and a post-state satisfying the class invariants and the postcondition, i. e.,

$$\forall \sigma \in V, p_1, \ldots, p_n.\ \sigma \models_{\text{pre}} (\text{pre}_{op}\ p_1 \ldots p_n)_{\text{pre}} \rightarrow \exists \sigma' \in V, \mathit{result}.\ (\sigma, \sigma') \models \text{post}_{op}\ p_1 \ldots p_n\ \mathit{result}$$

   for all public operations op with arguments $p_1, \ldots, p_n$ and with the return value *result*.

This notion is motivated by the following observations: if the condition described in item 1 is violated, there is always a method for which no argument can be passed that satisfies the invariants; if the conditions of item 2 or item 3 are violated, there is either no legal input or no function that maps it to output, i. e., the specification is not implementable. In all these cases, this means that the operation specification just means the empty transition relation which is semantically possible, but methodologically not desirable.

5.2 Proving Consistency of our Example

Recall our *abstract* model of a conference system presented in Section 2.1 (e. g., Figure 1) and assume that this model is defined in a package called `AbstractSimpleChair`. We start our consistency analysis by importing (and type-checking) the UML/OCL specification in HOL-OCL:

```
import_model "SimpleChair.zargo" "AbstractSimpleChair.ocl" include_only "AbstractSimpleChair"
```

This results in an environment holding all definitions and various automatically derived simplification rules of the data model defined in the `AbstractSimpleChair` package. In particular, this includes the class invariants for the classes `Person`, `Role`, `Participant` and `Session` as well as the specification of the operation `findRole`. We continue with the analytical command:

```
analyze_consistency [data_refinement] "AbstractSimpleChair"
```

which checks the syntactic requirements of our refinement methodology and, moreover, results in the generation of five proof obligations according to the schema described in the previous section. Each proof obligation is given an own name which can be used to process it. For example, if we *discharge* the second obligation resulting from the statement above, we can refer to it by:

```
po "AbstractSimpleChair.findRole_enabled"
```

The system reacts by changing to proof mode and displaying the assertions:

$$\exists \sigma \in V, self, P, R.\ \sigma \models_{\text{pre}} \neg\,(P \in self.\,\text{participants}_{\text{pre}})\,. \tag{51}$$

In proof mode, this assertion can be refined through backward-reasoning by a sequence of regular Isabelle proof commands or by specific HOL-OCL ones. The proof essentially consists in providing a witness for σ in form of an object graph with one `Person` and one `Session` object, where the participants list is just empty. Thus, the proof proceeds by establishing that $(\sigma, x) \models self.\,\text{participants}_{\text{pre}} \triangleq []$; the HOL-OCL simplifier will then complete the proof. After reaching the final proof state consisting of the formula true, one can state:

```
discharged
```

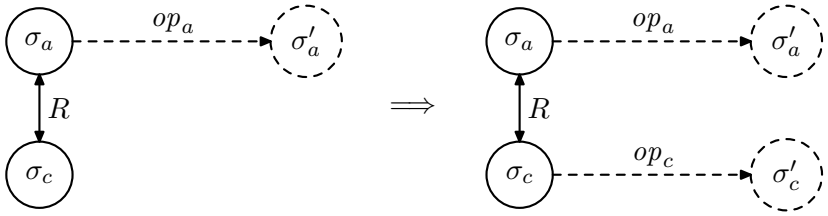
whereby this proof obligation will be erased from the database of proof obligations and added to the database of proven theorems.

5.3 Refining OCL Specifications

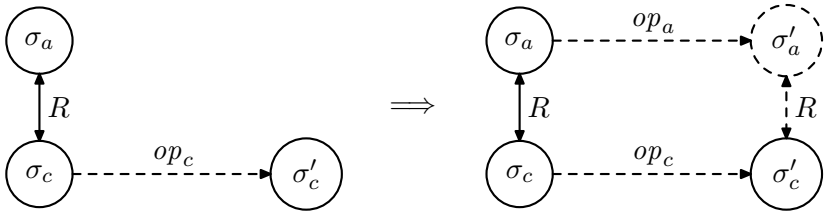
Data refinement is a well-known formal development technique; a standard-example for data refinement is Spivey's Birthday Book [31]. The key idea is to *refine* abstract, but easy-to-understand system models to more concrete, complex ones that are closer to an (executable) implementation. In prominent instances of the refinement method such as the B-Method, the final concrete model is converted to code via a trusted code-generator. According to a concrete formal refinement notion (such as *forward simulation* or *backward simulation*, c.f. [35]), stating that one model is a refinement of another one can be verified by checking syntactic constraints and by discharging (proving) automatically generated proof obligations.

Again, we will build our refinement method on the level of UML-packages: one containing the abstract model one containing the concrete model. We make the correspondence between abstract and concrete public classes and public operations on the basis of their name, i. e., classes or operations with same name correspond. This syntactic constraint allows for the direct substitutivity of the abstract package, i. e., in any place, where the specification requires the abstract package, we can also use the concrete one. To make refinements on packages semantically working, several side-conditions have to be imposed:

- the set of public classes of the abstract model must be *included* in the set of public classes of the concrete model;
- the set of public operations in a concrete class must be a subset of the public operations in the corresponding abstract class, and
- the types of the corresponding operations must match.



(a) Proof Obligation I



(b) Proof Obligation II

Figure 2 Proof obligations of a forward simulation refinement.

Refinement notions are typically based on putting the abstract states σ_a and concrete states σ_c into relation. To do this, an *abstraction relation* R must be provided by the user. An important special case is when R is in fact a function mapping concrete states to abstract states; although the proof obligations can be simplified in the functional case, we present the general case here. A forward simulation refinement $S \sqsubseteq^R_{FS} T \equiv po_1(S,R,T) \wedge po_2(S,R,T)$ comes in two parts which turn into proof obligations when stated as proof goals. They are best explained with a diagram, such as Figure 2. The first condition po_1 means that whenever an abstract operation S *can* make a transition, the corresponding concrete operation T can make a transition too. The second condition po_2 appears in Figure 2b. It states that whenever the concrete operation can make a step to a new system state σ'_c, then the abstract operation must be able to reach a state σ'_a that is in the abstraction relation to σ'_c.

To formalize these two conditions, two prerequisites are necessary that are related to the three-valuedness of the language:

$$\tau \models_M S \equiv (\tau \models S \vee \tau \models \neg\, \partial\, S) \tag{52}$$

and

$$\text{pre}\, S \equiv \{\sigma \in V \mid \exists \sigma' \in V.\ (\sigma, \sigma') \models_M S\}\,. \tag{53}$$

The former definition relaxes our notion of validity to "evaluating to true or to exception," which makes the exception view of $\perp$ explicit. The second definition characterizes the set of pre states in which an assertion S becomes valid. In these terms, the two proof obligations for an operation declared public in the abstract model can be expressed formally as follows:

$$po_1(S,R,T) \equiv \forall \sigma_a \in \text{pre}(S), \sigma_c \in V.\ (\sigma_a, \sigma_c) \in R \rightarrow \sigma_c \in \text{pre}(T) \tag{54}$$

and

$$\begin{aligned} po_2(S,R,T) \equiv \forall \sigma_a \in \text{pre}(S), \sigma_c \in V.\ \sigma_{c'}.\ (\sigma_a, \sigma_c) \in R \wedge (\sigma_c, \sigma'_c) \models_M T \\ \rightarrow \exists \sigma'_a \in V.\ (\sigma_a, \sigma'_a) \models_M S \wedge (\sigma_{a'}, \sigma_{c'}) \in R\,. \end{aligned} \tag{55}$$

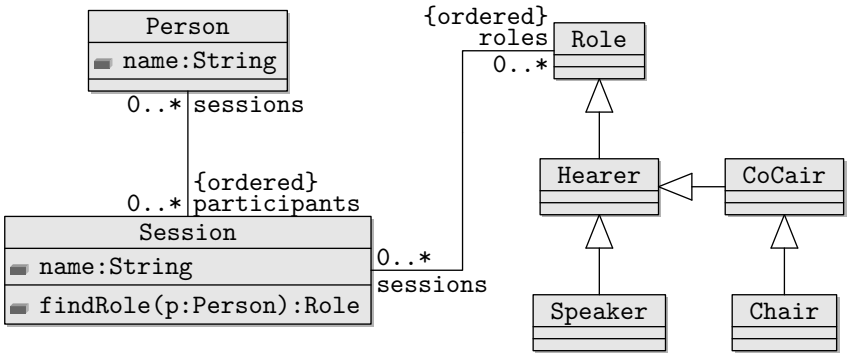


Figure 3 The concrete SimpleChair model avoids the use of association classes and, as such, is easier to implement in programming languages like Java.

However, these definitions leave open how to construct this global abstraction relation and how arguments of the operations are handled.

As a means to solve both problems, we suggest that the user provides a *family* of local abstraction relations R_C indexed by the *public classes* of the abstract model. Thus, we can relate input and result objects in the abstract state to corresponding objects in the concrete state. The global abstraction relation R can be constructed automatically by requiring that all abstract public objects can be associated "one-to-one" to concrete objects and that abstract objects relate to concrete objects with respect to a local abstraction relation R_C. There may be public objects in the concrete model that do not correspond to public objects in the abstract model.

5.4 A Brief Description of the Concrete SimpleChair Model

While the abstract version of the system is a "classical" data-model concentrating on data entities and its relations, such a model is difficult to implement; partly because high-level notations such as association classes are not supported directly by many tools, partly because a conversion to sequence attributes containing direct links to associated objects is more efficient, but more difficult since the state must be kept valid.

Figure 3 illustrates the class model of the concrete model that we define within the package `ConcreteSimpleChair`. The HOL-OCL specification differs mainly in the specification of the `findRole` operation which now uses the features of sequences.

```
context Session::findRole(person:Person):Role
  pre:  person ∈ self.participants
  post: result ≐ roles.at(participants.indexOf(p))
```

In contrast to the abstract variant, this specification is efficiently executable. Moreover, an additional invariant constraining the `Session` class describes that the sequences storing the roles and participants are of equal length:

```
context Session
  inv: ‖participants‖ ≐ ‖roles‖
```

The specification of the class `Person` remains unchanged:

```
context Person
  inv: name ≠ '' ∧ Person::allInstances()->isUnique(p:Person | p.name)
```

5.5 Proving Data Refinements of the SimpleChair-Example

We load the concrete model, analogously to the abstract model, into its own theory:

```
import_model "SimpleChair.zargo" "ConcreteSimpleChair.ocl" include_only ["ConcreteSimpleChair"]
```

Now we can import both theories into a refinement theory and declare the abstraction relations. This task is supported by the statement

```
refine "AbstractSimpleChair" "ConcreteSimpleChair"
```

of the HOL-OCL refinement component. The execution of the statement performs the following activities:

1. checking the syntactic side-conditions mentioned in Section 5.3,
2. declaring the local abstraction relation for the public classes, e. g., `Person`, `Role`, `Session`, of the abstract model,
3. constructing a predicate isPublic_a working for the objects of the data universe defined in the package `AbstractSimpleChair`,
4. constructing a predicate isPublic_c working for the objects of the data universe defined in the package `ConcreteSimpleChair`,
5. defining the global abstraction relation R (using the up-to-now undefined class abstractions), and
6. generating the refinement proof-obligations for the public operation `findRole`.

The motivation for the declaration of local class abstractions, which leave the definition to the user to a later stage, is a pragmatic one: giving the correct (HOL) type for an encoded HOL-OCL expression is usually quite sophisticated and requires experimenting in finding a suitable abstraction. For example, the definition that relates `Person` objects just relates objects with same `name` attribute:

$$\begin{aligned} R_{Person}\,\sigma_a\,\sigma_c\,obj_a\,obj_c \equiv \exists s.\ &(\sigma_a \models \text{AbstractSimpleChair.Person.name}\ obj_a \triangleq s) \\ &\wedge (\sigma_c \models \text{ConcreteSimpleChair.Person.name}\ obj_c \triangleq s)\,. \end{aligned} \tag{56}$$

Recall that the class invariant for `Person` requires that its objects are uniquely defined by their `name` attribute.

We now turn to the question of how to combine the family of local abstraction relations R_C to a global abstraction relation on states R. The core piece is the already mentioned requirement that there must be a one-to-one assignment between objects belonging to classes declared public in the abstract package. Furthermore, all assigned objects must be in the local abstraction relation, and the public visibilities must be preserved. Altogether, this is expressed as follows:

$$\begin{aligned} R\,\sigma\,\sigma' \equiv \exists f g.\ &\forall x \in \text{ran}\ \sigma.\ \text{isPublic}_a\ \sigma(K\,x) \rightarrow f(g\,x) = x \wedge \\ &\forall y \in \text{ran}\ \sigma'.\ \text{isPublic}_c\ \sigma'(K\,y) \rightarrow g(f\,y) = y \wedge \\ &\forall x \in \text{ran}\ \sigma.\ \text{isPublic}_a\ \sigma(K\,x) \rightarrow \text{isPublic}_c\ \sigma'(K(g\,x)) \wedge \\ &\forall x \in \text{ran}\ \sigma.\ \text{isPublic}_a\ \sigma(K\,x) \rightarrow R_{obj}\,\sigma\,\sigma'(K\,x)(K(g\,x)) \end{aligned} \tag{57}$$

where $K\,a = \lambda\,\sigma.\ a$ and isPublic_a and isPublic_c are *generated* predicates that decide if an object belongs to a public class in the abstract package. These predicates are just disjunctions of all dynamic type tests. Similarly, R_{obj} is a generated predicate combining the local abstraction relations by casting them appropriately to the common superclass, i. e., `OclAny`, and conjoining them disjointly. Finally, from the above refinement, two proof obligations arise expressing the refinement condition for each operation. The proof in HOL-OCL for the case of `findRole` proceeds as follows. We start by:

```
po "Refinement.findRole"
```

and get the display of:

$$\frac{\begin{array}{c}\forall\sigma \in \text{pre } S, \sigma' \in \text{pre } T.\ R_{Session}\ \sigma\ \sigma'\ self\ self' \\ \forall\sigma \in \text{pre } S, \sigma \in \text{pre } T.\ R_{Person}\ \sigma\ \sigma'\ p\ p' \\ \forall\sigma \in \text{pre } S, \sigma \in \text{pre } T.\ R_{Role}\ \sigma\ \sigma'\ result\ result'\end{array}}{\begin{array}{r}\text{AbstractSimpleChair.Session.findRole } self\ p\ result \\ \sqsubseteq^{R}_{FS}\ \text{ConcreteSimpleChair.Session.findRole } self'\ p'\ result'\end{array}}\ . \tag{58}$$

The three assumptions constrain the intended refinement relation to input and output parameters that are *representable* in the corresponding system state of the refining system. That is, for a person p in an abstract state, we must be able to relate it to a p'-object in the concrete state. This complication is a tribute to object-orientation: we cannot require, in a world of objects, that the arguments are simply equal as we could in a world of values. Rather, we must translate objects of one state to objects in another state to express the relation of object-graphs via its structure and not using the object-identifiers (references) that establishes it. Fortunately, since our example does not involve "deep" object graphs representing input of an operation to be refined, the local abstraction relations boil down to forgetting the object-id's and turning the person-objects into values (strings for names). In the general case, co-induction will be required. The rest of the 120 line proof is fairly straightforward and involves mostly the proof that whenever the abstract precondition is satisfied, the corresponding concrete precondition is also satisfied, as well as that the concrete postcondition is translatable into the abstract postcondition. This proof is also closed with:

```
discharged
```

6 Conclusion

6.1 Achievements

We presented a formal, machine-checked semantics of HOL-OCL as a conservative embedding into Isabelle/HOL. HOL-OCL strives for compliance with the UML/OCL standards, at least as far as the logic, its conception as an assertion language over object graphs, and the library types (except Set) are concerned. In some minor issues (as in the case of *infinite* sets), HOL-OCL generalizes the UML/OCL standard or makes it more precise (as in the case of *smashed* collections); in case of doubt, we opted for semantic definitions that resulted in simpler deductions.

On the basis of this conservative embedding, we derived several calculi and proof techniques for HOL-OCL. Since deriving means that we proved all rules within an interactive

theorem prover, we can guarantee both the consistency of the semantics as well as the soundness of the calculi. We developed automatic proof support for the derived calculi which are specialized to the language. In particular, the calculi led to rewriting and tableau-based decision procedures for certain fragments of HOL-OCL. The novel procedures have been applied for library development as well as medium-sized case studies.

We demonstrated the potential for applications of such deduction-based tools for the HOL-OCL system. We adopted classical analysis and data-refinement notion to three-valued, object-oriented HOL-OCL and showed how this can be used to relate specifications to implementations, even if based on different data universes. Thus, we provide a solid basis for turning object-oriented modeling using UML/OCL into a true formal method; at least as far as data-modeling aspects are concerned.

6.2 Related Work

We distinguish three areas of related work: general work on the UML/OCL and model-driven engineering, work on development by refinement and work on verification of object-oriented systems.

6.2.1 Model-driven Engineering

The term Model-driven Engineering (MDE) [18,30] refers to the systematic use of models as primary engineering artifacts throughout the development life-cycle of software systems. In the broadest sense, the term "model" is used for descriptions in a machine-supported format, while the term "systematic" refers to machine-supported transformations between models or from models to code. As a software development paradigm, MDE attracted interest in academia and industry.

HOL-OCL is in fact embedded in an MDE framework [7]. It consists of a *repository*, which is a database managing different versions of "models," an infrastructure to build model-transformations that has been used for the transformations of our running example in Section 5, and an experimental generic code-generator. Other model transformations described in [8] were used to transform security models described in SecureUML, a UML extension, together with a standard class model into a standard (secured) model. This transformation produces different proof-obligations; With the help of our framework, the combined model can be transformed to code, while the proof obligations making these transformations "correct" can be proven by HOL-OCL.

6.2.2 Refinement-oriented Development Methods

As a formal development method, the HOL-OCL approach is most closely related to the B-Method [1] and its most recent incarnation: Event-B [2]. The B-Method has been applied to substantial case studies of safety-critical systems.

In contrast to this tradition, HOL-OCL establishes a development method for object-oriented specifications and programs. Besides subtyping and inheritance, this means that formulae are assertions over a graph of objects linked via object identifiers. This introduces the technical complication that equality on values must be replaced by other user-defined equivalence relations, be it by using object-identifiers or recursive predicates representing bi-simulations.

6.2.3 Object-oriented Code-oriented Analysis Methods

Formal analysis of object-oriented systems has mostly been done in the context of code-verification. Systems like Boogie for Spec# [20], Krakatoa [22] or ESC/Java [21] for Java/-Java Modeling Language (JML) are using programming language code annotated by assertions. This is converted via a *wp*-calculus into proof-obligations which are handled by automated or interactive provers. While technically sometimes very advanced, the foundation of these tools is quite problematic: The generators usually supporting a large language are not verified, and it is not clear if the generated conditions are sound and complete with respect to the underling operational semantics. This turns out to be a particular problem if complex memory models are involved; The second author witnessed several stunning inconsistencies in these models for Boogie; for other systems, similar problems have been reported.

In contrast to these approaches, HOL-OCL generates a conservative model for objects and states (see [9,10]). Furthermore, HOL-OCL is geared towards top-down development via refinement, therefore complementary to these approaches. A first step towards code-verification via HOL-OCL is described in [10], where a Hoare-calculus for a small imperative language is derived.

A substantial body of literature on code-verification on object-oriented languages is based on deep embeddings in logical frameworks like Isabelle/HOL. Examples are embeddings of Java-Fragments [26], among them NanoJava [27], which focuses on meta-theoretic proofs like completeness. It served as a formal reference semantics in several other projects; however, complex side-condition hamper the efficiency of the reasoning considerably. While the approach is compatible to open world assumptions in principle, it is not easily amenable for modular verification.

Another code-based analysis tool is the KeY tool [3], which has a UML front-end and which is integrated into a state-of-the-art modeling tool; it is based on a two-valued version of dynamic first-order logic combined with a fragment of Java. KeY offers a rather powerful, specialized proof-procedure for large fragments of the language. In contrast to our conservative development, the library is just axiomatized. Methodologically, the approach is geared towards code-verification.

6.3 Future Work

6.3.1 Improving Technical Support

While our existing proof procedures for OCL are quite satisfactory, the overall efficiency needs to be increased and the a larger fragments of the language (including automated procedures for arithmetic, for example) should be covered. More configurations of our code-generator [7] are desirable for a wider range of examples.

6.3.2 Integration of Top-down and Bottom-up-Techniques

This paper explores the potential of HOL-OCL as a top-down development framework. It is our vision to integrate model MDE, development by refinement, code-verification and code-testing in one framework and thus to provide a combined semantical foundation as well as practical means for analysis of specifications.

6.3.3 Refinement

It is straightforward to integrate other refinement concepts into HOL-OCL, e. g., *backward simulation* $S \sqsubseteq^{R}_{\mathrm{BS}} T$ [35].

Finally, it is highly desirable to link method specifications to implementations in concrete code of a programming language like Java. Conceptually, this is a combination of the big-step OCL semantics with a Hoare Logic relating intermediate steps (cf. [34]). In [10], we present an implementation of this method within HOL-OCL; for space reasons, we have to refer the reader interested in formal proofs of this approach to the HOL-OCL distribution.

Acknowledgements We thank Lukas Brügger and Simon Meier for valuable discussions on the subject of this paper. Simon Meier implemented the described rewrite procedure.

References

1. Abrial, J.R.: The B-Book: assigning programs to meanings. Cambridge University Press, New York, NY, USA (1996)
2. Abrial, J.R.: Modeling in Event-B: System and Software Design. Cambridge, New York, NY, USA (2009)
3. Ahrendt, W., Baar, T., Beckert, B., Bubel, R., Giese, M., Hähnle, R., Menzel, W., Mostowski, W., Roth, A., Schlager, S., Schmitt, P.H.: The KeY tool. Software and Systems Modeling **4**(1), 32–54 (2005). doi: 10.1007/s10270-004-0058-x
4. Andrews, P.B.: Introduction to Mathematical Logic and Type Theory: To Truth through Proof, 2nd edn. Kluwer Academic Publishers, Dordrecht (2002)
5. Boulton, R., Gordon, A., Gordon, M.J.C., Harrison, J., Herbert, J., Tassel, J.V.: Experience with embedding hardware description languages in HOL. In: V. Stavridou, T.F. Melham, R.T. Boute (eds.) Proceedings of the International Conference on Theorem Provers in Circuit Design: Theory, Practice and Experience, IFIP *Transactions*, vol. A-10, pp. 129–156. North-Holland Publishing Co., Nijmegen, The Netherlands (1993)
6. Brucker, A.D.: An interactive proof environment for object-oriented specifications. Ph.D. thesis, ETH Zurich (2007). ETH Dissertation No. 17097.
7. Brucker, A.D., Doser, J., Wolff, B.: An MDA framework supporting OCL. Electronic Communications of the EASST **5** (2006).
8. Brucker, A.D., Doser, J., Wolff, B.: A model transformation semantics and analysis methodology for SecureUML. In: O. Nierstrasz, J. Whittle, D. Harel, G. Reggio (eds.) MoDELS 2006: Model Driven Engineering Languages and Systems, no. 4199 in Lecture Notes in Computer Science, pp. 306–320. Springer-Verlag (2006). doi: 10.1007/11880240_22.
9. Brucker, A.D., Wolff, B.: The HOL-OCL book. Tech. Rep. 525, ETH Zurich (2006).
10. Brucker, A.D., Wolff, B.: An extensible encoding of object-oriented data models in HOL. Journal of Automated Reasoning **41** (2008). doi: 10.1007/s10817-008-9108-3.
11. Church, A.: A formulation of the simple theory of types. Journal of Symbolic Logic **5**(2), 56–68 (1940)
12. Gabbay, D.M.: Labelled Deductive Systems, *Oxford Logic Guides*, vol. 1. Oxford University Press, Inc., New York, NY, USA (1997)
13. Gogolla, M., Richters, M.: Expressing UML class diagrams properties with OCL. In: T. Clark, J. Warmer (eds.) Object Modeling with the OCL: The Rationale behind the Object Constraint Language, *Lecture Notes in Computer Science*, vol. 2263, pp. 85–114. Springer-Verlag, Heidelberg (2002)
14. Hähnle, R.: Efficient deduction in many-valued logics. In: International Symposium on Multiple-Valued Logics (ISMVL), pp. 240–249. IEEE Computer Society, Los Alamitos, CA, USA (1994). doi: 10.1109/ismvl.1994.302195
15. Hähnle, R.: Tableaux for many-valued logics. In: M. D'Agostino, D. Gabbay, R. Hähnle, J. Posegga (eds.) Handbook of Tableau Methods, pp. 529–580. Kluwer Academic Publishers, Dordrecht (1999)
16. Jones, C.B.: Systematic Software Development Using VDM, 2nd edn. Prentice Hall, Inc., Upper Saddle River, NJ, USA (1990). 0-13-880733-7
17. Kerber, M., Kohlhase, M.: A tableau calculus for partial functions. In: Collegium Logicum—Annals of the Kurt-Gödel-Society, vol. 2, pp. 21–49. Springer-Verlag, New York, NY, USA (1996)
18. Kleppe, A., Warmer, J., Bast, W.: MDA Explained. The Model Driven Architecture: Practice and Promise. Addison-Wesley (2003)

19. Kobryn, C.: UML 2001: a standardization odyssey. Communications of the ACM **42**(10), 29–37 (1999). doi: 10.1145/317665.317673
20. Leino, K.R.M., Müller, P.: Modular verification of static class invariants. In: J. Fitzgerald, I.J. Hayes, A. Tarlecki (eds.) FM 2005: Formal Methods, *Lecture Notes in Computer Science*, vol. 3582, pp. 26–42. Springer-Verlag, Heidelberg (2005). doi: 10.1007/11526841_4
21. Leino, K.R.M., Nelson, G., Saxe, J.B.: ESC/Java user's manual. Tech. Rep. SRC-2000-002, Compaq Systems Research Center (2000).
22. Marché, C., Paulin-Mohring, C.: Reasoning about Java programs with aliasing and frame conditions. In: J. Hurd, T.F. Melham (eds.) Theorem Proving in Higher Order Logics (TPHOLS), *Lecture Notes in Computer Science*, vol. 3603, pp. 179–194. Springer-Verlag, Heidelberg (2005). doi: 10.1007/11541868_12
23. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL—A Proof Assistant for Higher-Order Logic, *Lecture Notes in Computer Science*, vol. 2283. Springer-Verlag, Heidelberg (2002). doi: 10.1007/3-540-45949-9
24. UML 2.0 OCL specification (2003). Available as OMG document ptc/03-10-14
25. Unified modeling language specification (version 1.5) (2003). Available as OMG document formal/03-03-01
26. von Oheimb, D.: Analyzing Java in Isabelle/HOL: Formalization, type safety and Hoare logic. Ph.D. thesis, Technische Universität München (2001)
27. von Oheimb, D., Nipkow, T.: Hoare logic for NanoJava: Auxiliary variables, side effects, and virtual methods revisited. In: L.H. Eriksson, P.A. Lindsay (eds.) FME 2002: Formal Methods—Getting IT Right, *Lecture Notes in Computer Science*, vol. 2391, pp. 89–105. Springer-Verlag, Heidelberg (2002). doi: 10.1007/3-540-45614-7_6
28. Richters, M.: A precise approach to validating UML models and OCL constraints. Ph.D. thesis, Universität Bremen, Logos Verlag, Berlin, BISS Monographs, No. 14 (2002)
29. Richters, M., Gogolla, M.: OCL: Syntax, semantics, and tools. In: T. Clark, J. Warmer (eds.) Object Modeling with the OCL: The Rationale behind the Object Constraint Language, *Lecture Notes in Computer Science*, vol. 2263, pp. 42–68. Springer-Verlag, Heidelberg (2002)
30. Schmidt, D.C.: Guest editor's introduction: Model-driven engineering. Computer **39**(2), 25–31 (2006). doi: http://doi.ieeecomputersociety.org/10.1109/MC.2006.58
31. Spivey, J.M.: The Z Notation: A Reference Manual, 2nd edn. Prentice Hall, Inc., Upper Saddle River, NJ, USA (1992)
32. Viganò, L.: Labelled Non-Classical Logics. Kluwer Academic Publishers, Dordrecht (2000)
33. Wenzel, M., Wolff, B.: Building formal method tools in the Isabelle/Isar framework. In: K. Schneider, J. Brandt (eds.) TPHOLS 2007, no. 4732 in Lecture Notes in Computer Science, pp. 351–366. Springer-Verlag, Heidelberg (2007)
34. Winskel, G.: The Formal Semantics of Programming Languages. MIT Press, Cambridge, Massachusetts (1993)
35. Woodcock, J., Davies, J.: Using Z: Specification, Refinement, and Proof. Prentice Hall International Series in Computer Science. Prentice Hall (1996).

A The Syntax of OCL

The OCL 2.0 standard uses a concrete syntax for OCL that is inspired by object-oriented programming languages. Whereas this textual notation is likely to be accepted easily by software developers, it looks unfamiliar and way too verbose for people with a formal methods background, in particular for proof engineers. Thus we developed a concise, more "mathematical" notation for OCL as an alternative, and used only the latter throughout the paper. Technically, both notations can be used in HOL-OCL.

Table 5 gives a brief overview over the translation table between both notations; the reader interested in a complete comparison may consult [6, 9].

Table 5: Different concrete syntax variants for OCL

	OCL (standard)	mathematical HOL-OCL
OclAny	`x = y`	$x \doteq y$
	`x <> y`	$x \not\doteq y$
	`o.oclIsUndefined()`	$\not\partial\, o$
	`o.oclAsType(t)`	$o_{[t]}$
	`o.oclIsType(t)`	$\mathrm{isType}_t\, o$
	`o.oclIsKindOf(t)`	$\mathrm{isKind}_t\, o$
	`t::allInstances()`	t`::allInstances()`
OclVoid	`OclUndefined`	$\bot$
	`o.oclIsUndefined()`	$\not\partial\, o$
	`o.oclIsDefined()`	$\partial\, o$
Integer	`x - y`	$x - y$
	`x + y`	$x + y$
	`x * y`	$x \cdot y$
	`x / y`	$x \,/\, y$
	`-x`	$-x$
Boolean	`true`	`T`
	`false`	`F`
	`x or y`	$x \vee y$
	`x and y`	$x \wedge y$
	`not x`	$\neg x$
	`x implies y`	$x \longrightarrow y$
	`if c then x else y endif`	`if` c `then` x `else` y `endif`
Collection	`X->size()`	$\lVert X \rVert$
	`X->includes(y)`	$y \in X$
	`X->count(y)`	X`->count(`y`)`
	`X->includesAll(Y)`	$X \subseteq Y$
	`X->isEmpty()`	$\emptyset \doteq X$
	`X->exists(e:T|P(e))`	$\exists\, e \in X\,.\, P(e)$
	`X->forAll(e:T|P(e))`	$\forall\, e \in X\,.\, P(e)$
Set	`Set{}`	$\emptyset$
	`X->union(Y)`	$X \cup Y$
	`X->intersection(Y)`	$X \cap Y$
	`X->complement(Y)`	X^{-1}