A Tool-supported Refinement Method for Object-oriented Data Models

Achim D. Brucker[1] and Burkhart Wolff[2]

[1] SAP Research, Vincenz-Priessnitz-Str. 1, 76131 Karlsruhe, Germany
achim.brucker@sap.com
[2] Universität des Saarlandes, 66041 Saarbrücken, Germany
wolff@wjpserver.cs.uni-sb.de

Abstract We present a tool-supported method for the analysis and refinement of object-oriented specifications using UML/OCL. Conceptually, our analysis is based on the proof of satisfiability of invariants and implementability of operations, while the refinement notion is an adoption of the well-known forward-simulation (as used in Z) to packages in UML/OCL.
The tool-support is essentially based on a generator of proof obligations, which is used both for the analysis of UML/OCL specification as well as for refinement conditions (for which syntactic side-conditions are checked). Both types of proof obligations can then be proven by automated and interactive techniques provided by the HOL-OCL proof environment.
Thus, we provide an integrated method for formal data refinement of object-oriented specifications.
Key words: Data Refinement, HOL-OCL, UML, OCL, Formal Method

1 Introduction

The transition from abstract to concrete system models, in particular to concrete code, via a formally defined and well-understood relation is known as *refinement*. Even if considering only data-oriented refinement methods, the body of literature is very substantial. Common approaches have been intensively discussed for formal methods such as Z [17] or B [1]. Moreover, some implementations [18, 9] of these methods have been used in relevant case-studies, demonstrating that refinement methods represent an effective alternative to code-verification.

On the one hand, many industrial modeling languages, like UML/OCL offer a very broad support for semi-formal modeling. On the other hand, they are often criticized for lacking a formally defined semantics and a tool-supported formal development process. Even though the first criticism is wide-spread, there are at least formal foundations for parts of the UML (e. g., class models annotated with OCL constraints). Nevertheless, it is true that implementations of formal analysis and refinement techniques are rare.

Generally speaking, a tool-supported refinement notion that is integrated into a UML/OCL-based Model-driven Engineering (MDE) process requires:

1. a tool-supported formal semantics for UML/OCL,

2. formal notions of consistency or well-formedness for UML/OCL,
3. formal refinement notions for UML/OCL, and
4. an integration of both into one framework.

In [5], we presented an MDE toolchain meeting the first and last requirement: built within an interactive theorem prover environment, HOL-OCL [6, 4] provides a formal semantic definition in HOL, interactive and automated proof-support as well as a model-repository and model-transformation environment for object-oriented specifications.

In this paper, we present a solution for the second and third requirement: based on a formal semantics for UML/OCL [4, 12], we define and implement within the HOL-OCL framework a formal analysis as well as a formal *data refinement* allowing to relate abstract class models to more concrete ones. Since our refinement notion is transitive, an original abstract design can be converted stepwise into a version that can be automatically converted into code.

The rest of this paper is structured as follows: after introducing the necessary preliminaries in Section 2, we present an approach for analyzing the consistency of a UML/OCL model in Section 3 which we see as a prerequisite for the data-refinement approach for UML/OCL specifications we present in Section 4. Finally, we draw conclusions and discuss related work in Section 5.

2 Background

2.1 UML/OCL by Example: The SimpleChair System

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 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 visualizes the underlying 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*). Moreover, classes can be grouped into packages, e. g., in our example all classes are part of the package `AbstractSimpleChair`.

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` 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

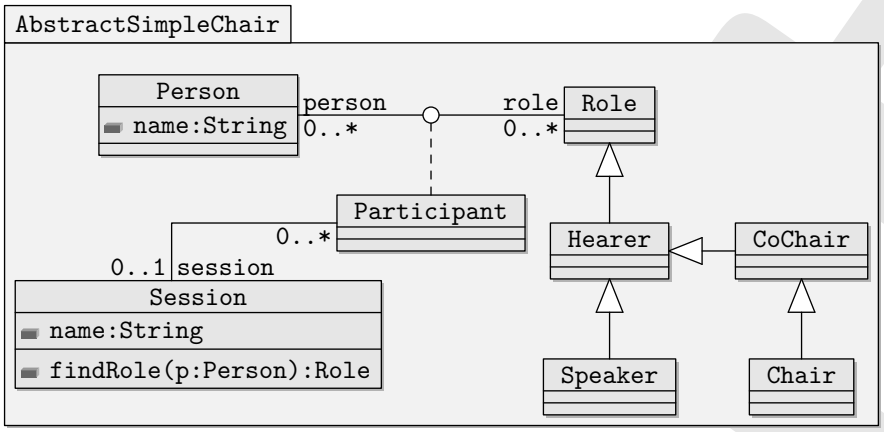


Figure 1. An abstract UML/OCL model of a simple conference management system. A person can participate in a session of a conference playing different roles, e. g., being a chair of a session or a speaker presenting a paper.

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 labeled 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 labeled 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.

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` is uniquely determined by the value of the `name` attribute:

```
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 $\text{isType}_{\texttt{Chair}}$ (`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. 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->select(p:Participant |
                                  p.person ≐ person ).role
        ∧ self.participants ≐ self.participants@pre
        ∧ self.name ≐ self.name@pre
```

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

In UML, classes can contain attributes of the type of the defining class. Thus, UML can represent (mutually) recursive datatypes, while OCL provides means to specify contracts of operations.

A key idea of defining the semantics of UML is to translate the diagrammatic UML features into a combination of more elementary features of UML and OCL expressions [10]. 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

The Meta-language Isabelle/HOL. higher-order logic (HOL) [8, 3] is a classical logic based on the typed λ-calculus, where quantifiers may range over arbitrary types. It has been implemented as instance in the generic proof-assistant Isabelle [11]; nowadays, Isabelle/HOL 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 $\_ \rightarrow \_$ which are constants of type $\text{bool} \Rightarrow \text{bool}$ or $[\text{bool}, \text{bool}] \Rightarrow \text{bool}$. Quantifiers are represented by higher-order abstract syntax; this means that $\forall\_.\_$ and $\exists\_.\_$ are usual constants of type $(\alpha \Rightarrow \text{bool}) \Rightarrow \text{bool}$ and that terms of the form $\forall(\lambda\, x.\, P\, x)$ are written $\forall x.\, P$. The Hilbert operator $\epsilon\, x.\, P$ returns an arbitrary x that makes $P\ x$ true. Further, there is the universal equality $\_ = \_$ of type $[\alpha, \alpha] \rightarrow \text{bool}$.

This core language can be extended via axiomatic ("conservative") definitions 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 $\alpha\,\mathsf{set}$ can be introduced isomorphic to the function space $\alpha \Rightarrow \mathsf{bool}$, 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 $\mathsf{dom}\ f$ and range $\mathsf{ran}\ f$ are defined. Moreover, on each type $\alpha_\perp$ a test for definedness is available via $\mathsf{def}\ x \equiv (x \neq \perp)$.

2.3 HOL-OCL

HOL-OCL [7, 6] (`http://www.brucker.ch/projects/hol-ocl/`) is an interactive proof environment for UML/OCL that is integrated into an MDE toolchain. Figure 2 shows this toolchain. HOL-OCL consists of:

- theories representing UML/OCL semantics as a conservative, shallow embedding into IsabelleHOL (following as closely as possible the standard [12]; in particular, HOL-OCL is based on three-valued, Strong Kleene Logic with laws like `true or OclUndefined = true`),
- libraries of derived rules based on this definition (as a consequence of the conservative methodology restricting ourselves to axiomatic definitions, we can guarantee their correctness and consistency),
- programmed proof-procedures for automated proof, and
- special support for UML/OCL class models which were automatically converted into logical theories representing their object-oriented datatypes.

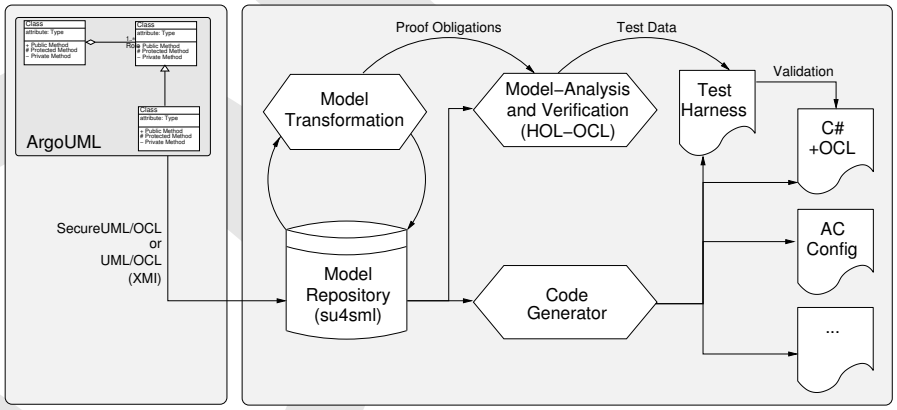


Figure 2. A MDE Framework and Toolchain integrating formal analysis allowing for a close relation between well-known MDE techniques (e. g., code-generation) and formal techniques (e. g., refinement).

Some fundamental remarks on the semantics are necessary here. The standard [12]) provides two semantic functions $I[\![P]\!]\ \sigma$ and $I[\![P]\!]\ (\sigma, \sigma')$ for OCL formulae P and states σ or σ'. These states are necessary to interpret the accessor functions (like `_.participants` or `_.participants@pre` in our example). Formulae containing no `@pre` accessors can be interpreted by both semantic functions. Since Isabelle/HOL uses overloading, the types will always provide an implicit distinction between them; Throughout this paper, however, we will use the variables τ, τ', τ'' for state-pairs, i. e., system transitions, to avoid confusion. Consequently, two notions of *validity* of OCL formulae are defined by:

$$\big(\sigma \vDash P\big) \equiv \big(I[\![P]\!]\ \sigma = \lfloor\!\lfloor \text{true} \rfloor\!\rfloor\big)$$
$$\big(\tau \vDash P\big) \equiv \big(I[\![P]\!]\ \tau = \lfloor\!\lfloor \text{true} \rfloor\!\rfloor\big),$$

i. e., a formula is *valid* if its semantic interpretation evaluates to true. Recall that the carrier set of the OCL type is three-valued and is interpreted by the three HOL values $\lfloor\!\lfloor \text{true} \rfloor\!\rfloor$,$\lfloor\!\lfloor \text{false} \rfloor\!\rfloor$ and $\bot$ of HOL-type $\text{bool}_{\bot}$.

The semantics of an operation $\mathit{self}.\, m_{op}(a_1, ..., a_n)$ is defined in the standard by the conjunction

$$\big(\sigma \vDash (\text{pre}_{op}\ \mathit{self}\ a_1 \ldots a_n)\big)\ \wedge\ \big((\sigma, \sigma') \vDash \text{post}_{op}\ \mathit{self}\ a_1 \ldots a_n\ \mathit{result}\big)$$

where pre_{op} and post_{op} denote the formula for the (syntactic) precondition or postcondition. Thus, similarly to the Z specification language, a collection of operations describes a state transition system.

3 Analyzing the Consistency of Class Models

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 desirable, but strictly speaking not required by the data refinement methodology we present later.

Most object-oriented languages support access specifiers. In UML this is called visibility which can be described as an enumeration ranging over private, protected, package, public. Whereas private members of a class are only accessible inside the class itself, protected parts are also accessible by subclasses, parts with visibility package are only accessible within the package and public members are accessible from everywhere. In contrast to many programming languages, e. g., Java, in UML also classes have a visibility. For example, a class with visibility package can only be used within the same package.

3.1 Formal Preliminaries

On the semantic side, we need constraints on states σ, not state transitions τ, therefore we need some formal machinery to switch between these two interpretations. Instead of using syntactic side-conditions (e. g., as in the OCL standard [12,

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 $(\sigma \vDash \phi) \overset{\text{pre}}{=} (\sigma \vDash \phi')$ formally as follows:

$$(\sigma \vDash \phi) \overset{\text{pre}}{=} (\sigma \vDash \phi') \equiv \forall x, y.\, \big((\sigma, x) \vDash \phi\big) = \big((\sigma, y) \vDash \phi'\big)\,,$$

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 refer to the pre-state (i. e., using `_@pre`) and where $\phi = \phi'$. Analogously, we define the concept of *post-state equivalence*, written $(\sigma \vDash \phi) \overset{\text{post}}{=} (\sigma \vDash \phi')$. Moreover, we introduce the notion *pre-state validity*:

$$(\sigma \vDash_{\text{pre}} \phi) \equiv (\sigma \vDash \phi) \overset{\text{pre}}{=} (\sigma \vDash \texttt{true})$$

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

Finally, we define a syntactic transformation $\_{}_{\text{pre}}$ of assertions to support the syntactical conventions of OCL. ϕ_{pre} results from ϕ by substituting all accessor functions by their `_@pre`-counterparts. Thus, we can express lemmas that link the standard semantic validity for invariants or preconditions (introduced as $\sigma \vDash \phi$ in Section 2.3) to formulae that can be interpreted within the validity of transitions:

$$\sigma \vDash \phi \Longrightarrow \sigma \vDash_{\text{pre}} (\phi)_{\text{pre}} \text{ and}$$
$$\sigma \vDash \phi \Longrightarrow \sigma \vDash_{\text{post}} \phi\,.$$

provided that in formula ϕ no `@pre` operator occurs. For formulae of this form (e. g., preconditions and invariants), the following property can also be proven automatically:

$$(\sigma \vDash_{\text{post}} \phi) \iff (\sigma \vDash_{\text{pre}} (\phi)_{\text{pre}})\,.$$

Based on these operations, we rephrase the notion of operation semantics for $\mathit{self}.\, m_{op}(a_1, \ldots, a_n)$ as follows:

$$\tau \vDash (\text{pre}_{op}\ \mathit{self}\ a_1 \ldots a_n)_{\text{pre}}\ \wedge\ \tau \vDash \text{post}_{op}\ \mathit{self}\ a_1 \ldots a_n\ \mathit{result}$$

This notion of operation semantics brings all semantic interpretation functions into a uniform format and therefore eases deduction:

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 $\lambda\,\text{oid}.\,\bot$ is always in V. Obviously, valid states enjoy the property:

$$\forall \sigma \in V; \mathit{self} \in \text{ran}\ \sigma.\, \sigma \models \text{inv}_C(\mathit{self})$$

where inv_C denotes the invariant of an arbitrary class C.

3.2 Formal Foundations of Model Consistency

Informally, we call a class model consistent if we can instantiate each class. Moreover, we require that the operations defined over the classes are executable in the sense that there exists a consistent post-state. While not strictly required by our refinement notions, we recommend a formal analysis showing the consistency of the overall class model in general and the packages that should be refined in particular.

Formally, we require for a consistent UML/OCL package:

1. For all (public) classes $C_1, \ldots, C_n$, there must exist a state satisfying the class invariants that contains objects of each class, i. e.,

$$\exists \{\sigma_1, \ldots, \sigma_n\} \in V, a_1 \in \operatorname{ran}\ \sigma_1, \ldots, a_n \in \operatorname{ran} \sigma_n. \\ \sigma_1 \vDash_{\text{post}} \operatorname{inv}_{C_1}(a_1) \wedge \cdots \wedge \sigma_n \vDash_{\text{post}} \operatorname{inv}_{C_n}(a_n)$$

   where $\operatorname{inv}_{C_1}, \ldots, \operatorname{inv}_{C_n}$ represent the class invariants.
2. For all (public) operations *op* of class C with arguments $p_1, \ldots, p_n$ of the class model, there must be a pre-state satisfying the class invariants and input variables satisfying the precondition, i. e.,

$$\exists \sigma \in V; \{self, p_1, \ldots, p_n\} \subseteq \operatorname{ran} \sigma. \sigma \vDash_{\text{pre}} (\text{pre}_{op}\ self\ p_1 \cdots p_n)_{\text{pre}}.$$

   This condition is also called the *enabling condition* in the literature.
3. For each valid pre-state all input variables satisfying the precondition, there must be a result and a post-state satisfying the class invariants and the postcondition, i. e.,

$$\forall \sigma \in V, \{self, p_1, \ldots, p_n\} \subseteq \operatorname{ran} \sigma. \sigma \vDash_{\text{pre}} (\text{pre}_{op}\ self\ p_1 \ldots p_n)_{\text{pre}} \\ \rightarrow \exists \sigma' \in V, result. (\sigma, \sigma') \vDash\ \text{post}_{op}\ self\ p_1 \cdots p_n\ result$$

   for all public operations *op* of class C with arguments *self*, $p_1, \ldots, p_n$ and with the return value *result*. This condition is also called *implementability* in the literature.

The first condition admits states, where not for *all* classes actually exist objects satisfying the invariants at the same time. Such systems exist in practice such that we were forced to relax the "intuitive" possibility "there exists a state with objects satisfying all invariants."

Overall, these proof-obligations ensure that each class can be instantiated (i. e., no invariant has an unsatisfiable class invariant) and that no operation specification describes the empty transition relation which is semantically possible, but methodologically not desirable.

3.3 Proving Consistency of our Example

To explain the first real HOL-OCL code-fragments, some general remarks are in place: HOL-OCL inherits from its underlying framework Isabelle [11] the concept of *hierarchical proof documents* that can be processed incrementally block

by block. In proof-documents (called theory files), blocks for declarations, definitions, axioms, documentation, SML-code, and proofs can be mixed arbitrarily. The input language of HOL-OCL is a super-set of the input language of Isabelle/HOL. Among others, HOL-OCL provides new commands for defining, analyzing, reasoning over object-oriented specifications.

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 package `AbstractSimpleChair`. 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 statement:

```
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:

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

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 $(\sigma, x) \vDash \mathit{self}.\,\text{participants}_{\text{pre}} \triangleq []$; 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.

4 Object-oriented Refinement

In this section, we present our object-oriented refinement method. We start with introducing the concrete version of our Simple Chair example (which refines

the abstract version). Thereafter, we present the two phases of our refinement notion: first we introduce the syntactical well-formedness checks and second, we introduce semantical foundation. We conclude this section with applying our refinement method to our running example.

4.1 Example: The Concrete SimpleChair Model

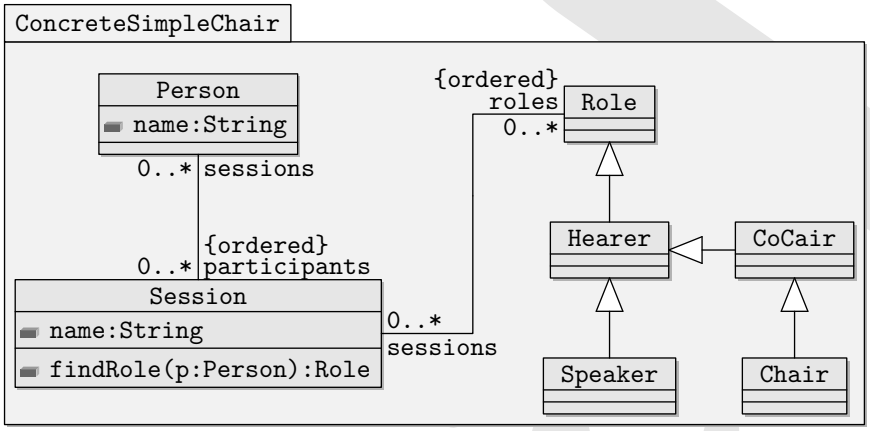


Figure 3. The concrete model of our simple conference management systems represents the participation directly in the `Session` class, omitting the private class `Participant`.

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)
```

4.2 Syntax and Well-Formedness

On the diagrammatic side, we use the UML notation, using stereotypes, for expressing refinement on the level of packages (see Figure 4). Our refinement

Figure 4. UML notation for refining on the level of packages.

method has both syntactic (also called *well-formedness* requirements) and semantic requirements. We discuss the former in this section.

The informal motivation for our refinement method is as follows: if package B refines a package A, then one should be able to substitute every usage of package A with package B. Thus, package B must at least provide all public operations (which define the signature of a package) that packages A provides. In more detail, we require

1. The concrete package must provide at a corresponding public class for each public class of the abstract model. For example, if the abstract package contains a class with name A, the concrete package must also contain a class with name A. As packages in UML define their own namespaces, these two classes can be distinguished by their fully qualified name.
2. For public attributes we require that their type and for public operations we require that the return type and their argument types are either basic datatypes (e. g., `Integer`, `String`) or public classes.
3. For each public class of the abstract package, we require that the corresponding concrete class provides at least
   (a) public attributes with the same name and
   (b) public operations with the same name.
   Moreover, we require that the types of corresponding abstract and concrete attributes and operations are compatible, i. e., either the same (e. g., in our running example, the attribute `name` of the abstract and concrete variant of the class `Session`, which is in both cases of type `String`) or are themselves in a refinement relation (e. g., in our running example, the return

type of the operation `findRole(...)` which returns in the abstract model the type `AbstractSimpleChair::Role` and `ConcreteSimpleChair::Role` in the concrete model).

Assuming that all classes of our running example, with the exception of the class `Participant` in the package `AbstractSimpleChair`, are public, is is quite obvious, that our example fulfills all well-formed requirements mentioned above.

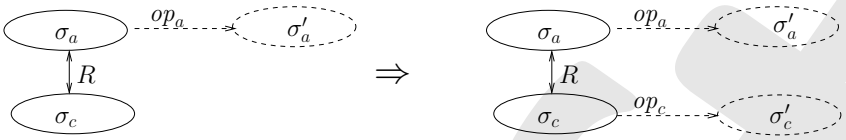
4.3 Refining OCL Specifications

Data refinement is a well-known formal development technique; a standard-example for data refinement is Spivey's Birthday Book [17]. 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. [19]), stating that one model is a refinement of another can be verified by checking syntactic constraints and by discharging (proving) the generated proof obligations.

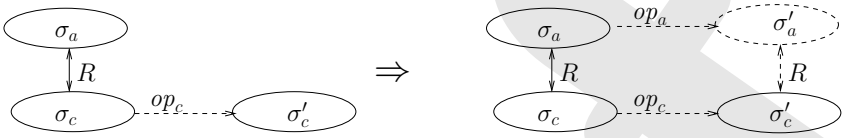
Again, we will build our refinement method on the level of UML-packages: one containing the abstract model, another one the concrete model. We make the correspondence ("matching") between abstract and concrete public classes and public operations on the basis of their name, i. e., classes or operations 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.

Refinement notions are typically based on putting the abstract states σ_a and concrete states σ_c into relation. This relation is defined by an *abstraction relation* R which 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 5. The first condition po_1 (see Figure 5a) 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 5b. It states that whenever the concrete operation can make a step to a new 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.



(a) Proof Obligation I: whenever an abstract operation S *can* make a transition, the corresponding concrete operation T can make a transition too



(b) Proof Obligation II: 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.

Figure 5. Showing a forward simulation refinement relations requires results in two proof obligations that need to be shown.

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

$$\tau \vDash_M \ S \equiv (\tau \vDash \ S \vee \tau \vDash \ \neg \partial \ S)$$

and

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

The former definition relaxes our notion of validity to "evaluating to true or to exception," which makes the exception view of $\bot$ 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:

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

and

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

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.

4.4 Proving Data Refinements of the SimpleChair-Example

We load the concrete model, analogously to the abstract model, into its own HOL-OCL 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 4.2,
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 abstract model, i. e., the `AbstractSimpleChair` data universe,
4. constructing a predicate isPublic_c working for the objects of the concrete model, i. e., the `ConcreteSimpleChair` data universe,
5. defining the global abstraction relation R (using the up-to-now undefined class abstractions), and
6. generating the refinement related proof-obligations for the public operations `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:

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

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 public-ness must be preserved. Altogether, this is expressed as follows:

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

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. Conceptually, we need to show the following lemma for `findRole`:

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

Within HOL-OCL, the proof attempt would be started by

```
po "refine_findRole"
```

which opens a proof state requiring a proof for the refinement of the operation `findRole`. This proof obligation is then shown by a sequence of interactive proof steps. After the successful completion of the proof, we can close our proof attempt and thereby mark this proof obligation as proved with the command:

```
discharged
```

Table 1 depicts a sketch (due to space reasons we skipped the actual proofs that po_1 and po_2 hold, i. e., this sketch only describes the high-level structure of the refinement proof) of such an interactive proof showing that our example is a valid refinement, i. e.., the refinement related proof obligations hold. This example should give a flavor how proofs of this form look like: 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

```
po "refine_findRole"
this opens a proof state requiring a poof for:
assumes Session_relates : ∀ τ τ'.
        R_AbstractSimpleChair_ConcreteSimpleChair_Session τ τ' self self'
assumes Person_relates : ∀ τ τ'.
        R_AbstractSimpleChair_ConcreteSimpleChair_Person τ τ' p p'
assumes Role_relates : ∀ τ τ'.
        R_AbstractSimpleChair_ConcreteSimpleChair_Role τ τ' result result'
shows
  refine (AbstractSimpleChair.Session.findRole self p result)
         (R_glob_St is_public_abs is_public_conc
                R_AbstractSimpleChair_ConcreteSimpleChair_obj_St)
         (ConcreteSimpleChair.Session.findRole self' p' result')
  we start the proof by unfolding the definitions of therefinement related
  proof obligations, i. e.., refine, po1, and po2:
proof(auto simp: refine_def po1_def po2_def)
  the first case shows po1, i. e., findRole reflects enabledness.
  case goal1 then show ?case
          apply(auto simp: refinement_simpset)
          due to space reasons, we apply a previously proven lemma that po1 holds.
          apply(rule findRole_po1)
          apply(simp_all add: findRole_conc_spec)
          done
  the shows case shows po2, i. e., findRoleproduces states in refinement relation.
  case goal2 then show ?case
          apply(auto simp: refinement_simpset)
          apply(frule find_Role_reads_only)
          du to space reasons, we apply a previously proven lemma that po2 holds.
          apply(rule findRole_po2)
          apply(auto simp: findRole_conc_spec)
   done
   next
discharged
```

Table 1. A skeleton showing the main proof steps needed for proving data refinement within HOL-OCL: after unfolding the main definitions, the proof can be split into two cases, one showing po_1 and one showing po_2.

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). Overall, the 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.

5 Conclusion and Related Work

In this section, we summarize related work, draw conclusions and give a short outlook on future work.

5.1 Related Work

[13, 14] presents a OCL-based refinement notions for UML classes which in fact is based on the refinement notions for Object-Z [15]. Both approaches are only discussing the refinement conditions for class-wise refinement, i. e.. a class refines another one. Moreover, neither of these approaches supports the formal verification of the refinement within an integrated MDE environment: [13, 14] describes a validation approach using simulation and Object-Z is supported by generic analysis tools, e. g., [16], that do not provide specialized refinement support.

In contrast, our approach provides a refinement methodology (including well-formedness checks) for UML packages. Moreover, it is integrated into a formal MDE environment supporting the well-formedness checks, the generation of refinement conditions, and their formal analysis in a theorem prover.

As such, HOL-OCL [7] as a methodology is most closely related to the B-Method [1] and its most recent incarnation: Event-B [2]. Both variants of the B-Method method are centered around the idea of having formal, tool-supported refinement notion.

With our work, we try to transfer this setting to 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 also leads to additional challenges, for example, the equality on values must be replaced by other user-defined equivalence relations, be it by using object-identifiers or recursive predicates representing bi-simulations. Thus, compared to the non-object-oriented B Method, refinement proofs for object-oriented systems are substantial more complicated, and, thus, require more advance tool support.

5.2 Conclusion and Future Work

In this paper, we presented a formal refinement methodology for object-oriented specifications. Our object-oriented refinement methods support the refinement

on the level of packages, i.e., it generalizes class-wise refinement approaches, e. g., [13, 16]. Moreover, both the syntactic well-formedness checks required by our refinement notions and the generation of semantic proof obligations is supported by a tool. In fact, our refinement methodology is seamlessly integrated into an MDE toolchain [5].

We see several lines for future work, aiming mainly on increasing the overall usability and usefulness of our tool support for object-oriented refinement. This comprises, among others,

- increase specialized proof support for proving refinement conditions, i.e particular variants for functional refinement relations or systems with more liberal object-relations that "one-to-one,"
- integrate fully automatic proof attempts into the proof-obligation generator,
- provide automatic heuristics for defining the refinement relations, and
- relaxing the syntactical requirements by providing means for defining a more flexible refinement relation.

References

[1] J.-R. Abrial. *The B-book: assigning programs to meanings.* Cambridge University Press, New York, NY, USA, 1996.
[2] J.-R. Abrial. *Modeling in Event-B: System and Software Design.* Cambridge, 2008. To appear.
[3] P. B. Andrews. *Introduction to Mathematical Logic and Type Theory: To Truth through Proof.* Kluwer Academic Publishers, Dordrecht, 2nd ed., 2002.
[4] A. D. Brucker. *An Interactive Proof Environment for Object-oriented Specifications.* Ph.d. thesis, ETH Zurich, 2007. ETH Dissertation No. 17097.
[5] A. D. Brucker, J. Doser, and B. Wolff. An MDA framework supporting OCL. *Electronic Communications of the EASST*, 5, 2006.
[6] A. D. Brucker and B. Wolff. The HOL-OCL book. Tech. Rep. 525, ETH Zurich, 2006.
[7] A. D. Brucker and B. Wolff. HOL-OCL – A Formal Proof Environment for UML/OCL. In J. Fiadeiro and P. Inverardi, eds., *Fundamental Approaches to Software Engineering (FASE08)*, no. 4961 in Lecture Notes in Computer Science, pp. 97–100. Springer-Verlag, 2008.
[8] A. Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5(2):56–68, 1940.
[9] Clearsy Inc. Atelier B, 2008. `http://www.atelierb.eu/`.
[10] M. Gogolla and M. Richters. Expressing UML class diagrams properties with OCL. In T. Clark and 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.
[11] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL—A Proof Assistant for Higher-Order Logic*, *Lecture Notes in Computer Science*, vol. 2283. Springer-Verlag, Heidelberg, 2002.
[12] UML 2.0 OCL specification. 2003. Available as OMG document ptc/03-10-14.
[13] C. Pons and D. Garcia. An OCL-based technique for specifying and verifying refinement-oriented transformations in MDE. In O. Nierstrasz, J. Whittle, D. Harel, and G. Reggio, eds., *MoDELS*, *Lecture Notes in Computer Science*, vol. 4199, pp. 646–660. Springer-Verlag, Heidelberg, 2006.

[14] C. Pons and D. Garcia. Practical verification strategy for refinement conditions in UML models. In S. F. Ochoa and G.-C. Roman, eds., *IFIP Workshop on Advanced Software Engineering*, *IFIP*, vol. 219, pp. 47–61. Springer, 2006.

[15] G. Smith. *The Object Z Specification Language*. Advances in Formal Methods Series. Kluwer Academic Publishers, Dordrecht, 2000.

[16] G. Smith, F. Kammüller, and T. Santen. Encoding Object-Z in Isabelle/HOL. In D. Bert, J. P. Bowen, M. C. Henson, and K. Robinson, eds., *ZB 2002: Formal Specification and Development in Z and B*, *Lecture Notes in Computer Science*, vol. 2272, pp. 82–99. Springer-Verlag, Heidelberg, 2002.

[17] J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall, Inc., Upper Saddle River, NJ, USA, 2nd ed., 1992.

[18] M. Wenzel and B. Wolff. Building formal method tools in the Isabelle/Isar framework. In K. Schneider and J. Brandt, eds., *TPHOLS 2007*, Lecture Notes in Computer Science, pp. 351–366. Springer-Verlag, Heidelberg, 2007.

[19] J. Woodcock and J. Davies. *Using Z: Specification, Refinement, and Proof*. Prentice Hall International Series in Computer Science. Prentice Hall, 1996.