Model Transformation as Conservative Theory-Transformation

Achim D. Brucker[*], **Frédéric Tuong**[†], **and Burkhart Wolff**[§]
[*]University of Exeter, UK
[†]Trinity College Dublin, Ireland
[§]Université Paris-Saclay, France

ABSTRACT Model transformations play a central role in model-driven software development. Hence, logical unsafe model transformation can result in erroneous systems. Still, most model transformations are written in languages that do not provide built-in safeness guarantees.
We present a new technique to construct tool support for domain-specific languages (DSLs) *inside* the interactive theorem prover environment Isabelle. Our approach is based on modeling the DSL formally in higher-order logic (HOL), modeling the API of Isabelle inside it, and defining the transformation between these two. Reflection via the powerful code generators yields code that can be integrated as extension into Isabelle and its user interface. Moreover, we use code generation to produce tactic code which is bound to appropriate command-level syntax.
Our approach ensures the logical safeness (conservativity) of the theorem prover extension and, thus, provides a *certified* tool for the DSL in all aspects: the deductive capacities of theorem prover, code generation, and IDE support. We demonstrate our approach by extending Isabelle/HOL with support for UML/OCL and, more generally, providing support for a *formal object-oriented modeling method*.

KEYWORDS Model Transformation; Conservativity; UML; OCL; Isabelle/HOL.

1. Introduction

Model transformations play a central role in model-driven software development. Hence, logical unsafe model transformation can result in erroneous systems. Still, most model transformations are written in languages that do not provide built-in safeness guarantees. In contrast, logically safe extensionality has been a key-feature of interactive theorem proving systems in the higher-order logic (HOL) family. The main goal of these systems (inspired by the LCF system) is to achieve *correctness by construction* for primitive inferences in a fairly small kernel, combined with flexible programmability in user space. The implementation language SML (**?**) protects the inference kernel by its type discipline, and top-level command interaction allowing for the development of layers of commands over this kernel. Extensionality is provided in modern systems like Coq (**?**) or Isabelle (**?**) by user-friendly high-level languages such as Gallina (**?**) for Coq or Isar (**?**) for Isabelle.

Extensionality leverages the scalability of the definitional principles of the LCF approach, paving the way for specific support of specification constructs for, e. g., datatypes or recursive function definitions. Specification constructs, are short-hands for *theory extensions* consisting of a collection of constant and type declarations, definitional axioms, and tactic proofs establishing proofs for a number of derived rules to reason over this construct (see Section 2 for more details).

A theory is a pair (Σ, Φ) of a signature Σ and a set of formulas Φ (denoting axioms or theorems), a *theory extension* is a map $(\Sigma, \Phi) \mapsto (\Sigma \uplus \Sigma', \Phi \uplus \Phi')$. A theory extension is *conservative* iff in the resulting super-theory no new theorems can be proved about the language of the original theory. As a consequence, logically consistency (falsehood not derivable) of the original theory is preserved in the super-theory.

Logically safe extensionality (or synonymous: *conservativity* (**?**)) is at the heart of this paper. We argue that in the context of semantics for domain-specific languages (DSLs), *conservativity* is a more relevant notion than *correctness*. This is because

This is the author's version of the work. It is posted at https://www.brucker.ch/bibliography/abstract/brucker.ea-model-2020 for your personal use. Not for redistribution. The definitive version was published in *Journal of Object Technology*, pp. 1–15, 2020, CC BY-ND.

the notion of correctness involves the following ingredients:

- a translation function T mapping the DSL to some target language L,
- a common semantic domain D,
- two semantic functions I_{DSL} and I_L, and,
- a proof that T preserves the underlying semantics, i.e., $I_L(T(X)) = I_{DSL}(X)$.

The notion of correctness implies obviously a lot of formal machinery which is in stark contrast to the simplicity and attractivity of the DSL approach: simply define a new language with appropriate syntax in terms of a language that one knows, and that has tool support. However, if the target language L is a specification language, or even a general logic like HOL, as in our case, the critical question is whether the result of the translation $T(d)$ is logically consistent.

Compiling an entire DSL definition into a series of conservative specification constructs and deriving the resulting DSL-specific rules and algebras automatically is a dauntingly complex task. The main objective of this paper is to present a *technique* to reduce this complexity and to demonstrate the feasibility of this technique on a substantial case study. Instead of writing a DSL-to-theory compiler in SML, we propose to write it logically in Isabelle/HOL itself, to use Isabelle's code generator to convert it into SML, and to bind the resulting code to specific command-level syntax of the Isabelle system. The generated DSL tool reuses the infrastructure of the theorem prover platform, such as the Prover IDE (PIDE), the code generator and the documentation generation facilities. Last but not least, the DSL tool possesses integrated automated and interactive proof support, from which the developer can profit in each stage of its development process.

In more detail, our technique comprises:

1. An (abstract-syntax) model of the Isabelle API. This model has been published in (**?**) and can be reused by developers of other DSL support extensions.

2. An (abstract-syntax) model of UML/OCL. This model has been published in (**?**), together with a functional working example in (**?**).

3. A compiler written in HOL mapping class diagrams to Isabelle/HOL definitions and Isabelle/Isar proofs.

Thus, similar to specification constructs, a component is built that derives the lemmas of an "object-oriented datatype theory" from a class model. Being the basis for more abstract proofs from the problem domain, they allow for formal code verification, refinement and test-generation techniques that UML models usually lack.

We showcase our technique by building a certified proof tool for a fragment of UML/OCL that focuses on the contract language for data invariants and operations.

The rest of the paper is structured as follows: after introducing the background of our work (Section 2), we present our tool (Section 3). Then, we discuss the formal semantics (Section 4) of UML/OCL, followed by our approach to support a DSL as a certified extension (Section 5). Next, we illustrate the resulting tool (Section 6) and discuss our lessons learned from following two different implementation strategies for building a formal UML/OCL tool based on Isabelle/HOL (Section 7). Finally, we draw conclusions in Section 8.

2. Background: Isabelle and UML/OCL

In this section, we briefly outline the architecture of Isabelle as well as the logic HOL (our semantic meta-language). Then we introduce UML/OCL (the DSL, in our example) as a formal specification language.

2.1. The Isabelle system architecture

The implementation of our technique has been realized on the Isabelle system; to discuss the issues of "certification" and "trust," it is helpful to give an overall glimpse on its architecture (see Fig. 1). The system is built upon an SML (**?**) interpreter and compiler. SML is a strongly typed language with a powerful module system. On this layer resides the system-kernel implementing *types* and λ-*terms*, *signatures* and *theories*. The particular type `thm` consists basically of triples of the form $\Gamma \vdash_\theta \phi$, stating that the Boolean term ϕ could be derived in theory θ from the assumptions Γ, i. e., a list of Boolean terms. The `thm` type is protected by the SML module system, which means that only kernel operations from this layer can produce `thm` objects; all higher layers providing tactics and packages reside on these operations of this fairly small kernel. These characteristics of a small and well protected kernel are common for the family of LCF-style systems such as Coq, HOL4, HOL-light, or Isabelle. The *package* layer provides support for specification constructs such as constant and type definitions, inductive datatypes, total recursive function definitions etc, which give sufficient user support for common modeling situations.

2.2. Basic conservative specification constructs

We present two very basic specification constructs to explain the underlying principles:

1. *Constant definitions*, written as follows in Isabelle's input language Isar:

   Isabelle (Isar)
   ```
   definition c::τ where c =  E
   ```

   Constant definitions can be seen as a kind of macro for the declaration of a constant symbol `c` of type τ, and an axiom `c = E` (where `E` is an expression). However, a number of syntactic checks turn it into a *definitional* axiom. Most notably: `c` must not be defined already, `c = E` must be well typed, `E` must not contain free variables, and `c` must not occur in `E` (i.e, no recursion). These syntactic constraints are mechanically checked. It is not difficult to see that a definition works like the introduction of an abbreviation and is thus conservative: in all proofs of the super-theory, the constant symbol `c` can be expanded which gives a proof in the original theory.

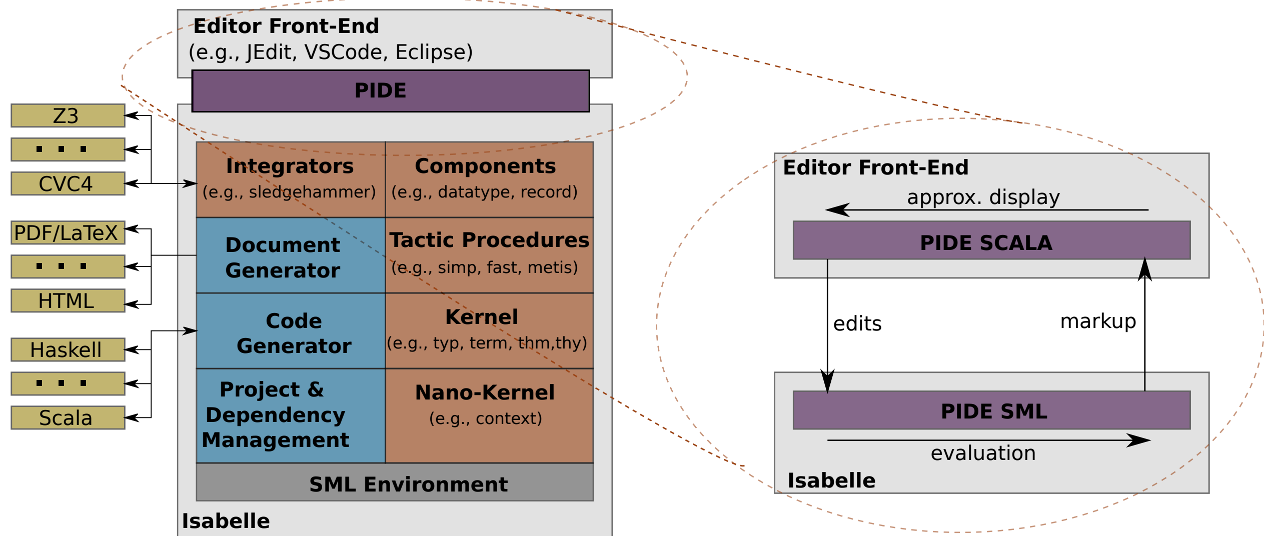


Fig. 1 The system architecture of Isabelle (left-hand side) and the asynchronous communication between the Isabelle system and the IDE (right-hand side).

2. *Type definitions*, written as follows:

Isabelle (Isar)
```
typedef τ = E
```

Type definitions follow the same idea as constant definition: a new type is a kind of abbreviation for an old one. Since for any type τ one can construct a characteristic function of type $\tau \Rightarrow$ `bool`, it is straightforward to construct a typed set theory inside HOL via conservative definitions, providing the type τ `set`. Now, a *type definition* τ = `E` constructs an isomorphism between a non-empty subset of elements characterized by `E` and the fresh type symbol τ. This isomorphism is represented via two fresh constant symbols and two axioms linking them to the subset `E`. Isabelle generates a proof obligation requiring that the set denoted by `E` is non-empty.

All higher-level specification constructs of Isabelle, such as datatype definitions, record definitions, or definitions of (well-founded) recursive functions, are based on these two constructions. Actually, the entire HOL library constructed over the axiomatized core logic. Consequently, logical consistency of the entire library boils down to the consistency of the core logic, which is generally accepted; for example, see (?) for a model of the core of HOL in ZFC (?).

2.3. Isabelle's HOL library

The HOL library theories comprise a typed set theory; it supports, e. g., the usual notation `{x. P(x)}` for set comprehensions or `x ∈ S` for set membership. We use the datatype specification construct to define new datatypes such as the `option` type, which is similar to the `Maybe` type in Haskell:

Isabelle (Isar)
```
datatype
  'a option = None | the:Some 'a
```

This introduces the usual constructors `None` and `Some` on this implicitly declared `option` type. It defines also the selector `the` and derives the lemma `the(Some X) = X`.

The type of partial functions is `'a ⇒ 'b option`, usually denoted by `'a ⇀ 'b` (technically, `'a ⇀ 'b` is a type synonym for `'a ⇒ 'b option`). The construct also automatically *generates* proofs that establish a number of rules resulting from these definitions such as the distinctness of `Some` and `None`, the injectivity of `Some` and an induction principle.

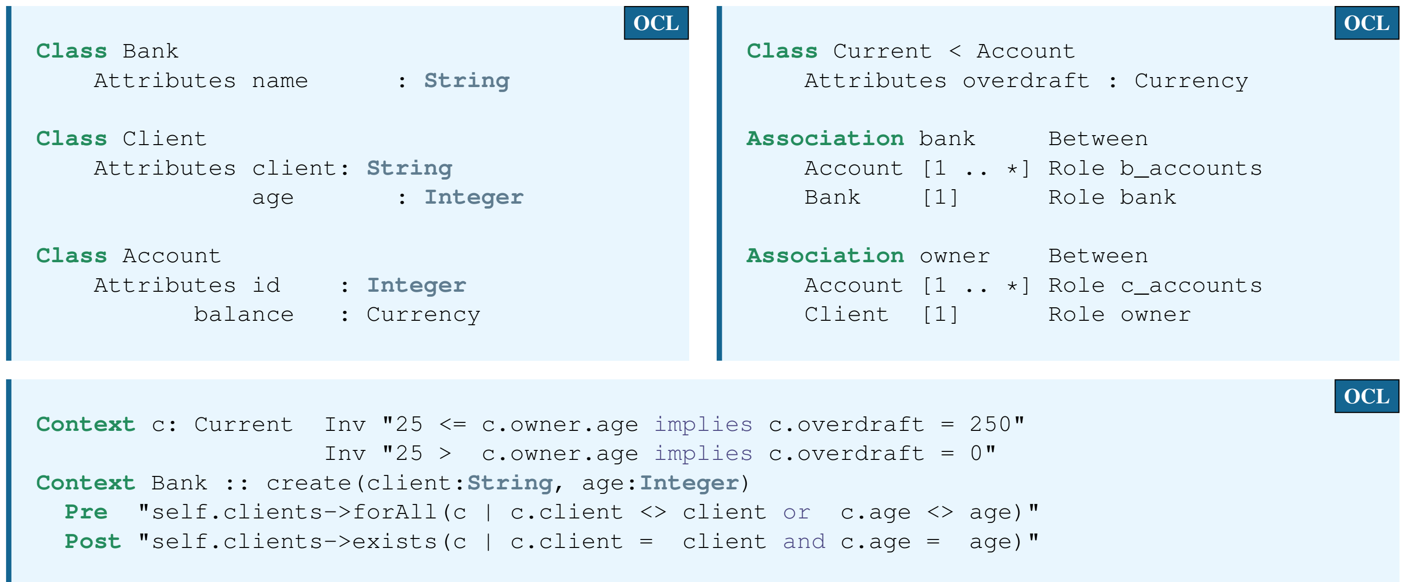
Finally, the Isabelle library comprises conservative theories for lists, pairs, total and partial functions, and arithmetic.

2.4. A guided tour through UML/OCL

UML (?) and its textual extension OCL (?) are one of the few industrially used modeling-languages. While UML class models mostly declare types and data occurring in a state (i. e., *objects*), OCL expressions constrain the set of states and state transitions. In the following, we will introduce UML/OCL by a small example of a class model together with its class invariants and a method contract in OCL. Fig. 2 describes a set of clients owning bank accounts using a textual representation that we share with USE (?).

In our example, each account is either a "current account" or a "saving account" (specified with the *inheritance relation* "_ < _"), and belongs to exactly one bank and one client. The relation between an `Account` and a `Bank` is modeled as an *association* with *multiplicity* constraints that describe, e. g., if the relation is of type "one-to-one" (and, thus, bijective), "one-to-many" (and, thus, injective), or "many-to-many."

For expressing more complex *data invariants* as well as *operation contracts*, the UML can be enriched with the Object Constraint Language (OCL) (?). OCL is a textual language; its



```
Class Bank
    Attributes name      : String

Class Client
    Attributes client: String
               age       : Integer

Class Account
    Attributes id     : Integer
           balance    : Currency
```

```
Class Current < Account
    Attributes overdraft : Currency

Association bank     Between
    Account [1 .. *] Role b_accounts
    Bank    [1]      Role bank

Association owner    Between
    Account [1 .. *] Role c_accounts
    Client  [1]      Role owner
```

```
Context c: Current  Inv "25 <= c.owner.age implies c.overdraft = 250"
                    Inv "25 >  c.owner.age implies c.overdraft = 0"
Context Bank :: create(client:String, age:Integer)
  Pre  "self.clients->forAll(c | c.client <> client or  c.age <> age)"
  Post "self.clients->exists(c | c.client =  client and c.age =  age)"
```

Fig. 2 A simple class model with OCL constraints capturing a bank account.

core logic is a four-valued logic that contains two exceptional elements: `null` and `invalid`. Here, `null` is a *non-strict* element that represents, similar to the null reference in object-oriented programming languages, the absence of a value. In contrast, `invalid` is a *strict* exception element representing undefined behavior such as the result of a division by zero.

In our example, we express the fact that clients of age 25 or older are allowed an overdraft up to € 250 as an OCL invariant of the class `Current` account. Moreover, we describe the semantics of the constructor the class `Bank` as a operation contract, i. e., a pair of pre- and postconditions.

The OCL expression language only appears inside the invariants and the pre- and post conditions. It provides basic operations such as the logical connectives such as `and`, `implies` as well as the arithmetic operations `_ <= _` and finally operations for sets and sequences of data,e. g., `forAll`. The OCL expressions above contain operation symbols that were induced by the class model above; for example, the operation `.owner` is implicitly declared in the association `owner` and maps to each object of (sub-)class `Account` the set of `Client`'s which are in the `owner`-relation to this object; due to the cardinality constraints this set is known to have exactly one element. Furthermore, the operation `.age` is implicitly declared in the class definition `Client` and projects to a given `Client` object the content of the `age` attribute. The semantics of the operations, such as `.owner` and `.age`, depends on the given class model; it is the Isabelle theory of these operations that constitutes the "object-oriented datatype theory" of the above class model.

3. A certified proof tool for OCL

A formal methods tool for the fragment of UML/OCL comprising the contract-language for data invariants and operations and thus supporting object-oriented data-modeling consists of two components:

1. a formal semantics of the basic OCL operations (i. e., logic, arithmetic, collection types); the semantics of the OCL library is given as shallow embedding into HOL, and
2. a theorem prover extension that accepts a textual representation of a class model and converts it into an object-oriented datatype theory. It should reflect the typing discipline of UML/OCL tightly and be certifying.

The resulting system is integrated into Isabelle, supporting the usual IDE-like editing, exploration, document and code generation facilities of the Isabelle system. This is possible due to the flexibility of Isabelle's Isar language (**?**) and its IDE Isabelle/jEdit (**?**).

The screenshot in Fig. 3 gives a glimpse of the result of our entire construction: the end-user interface of our tool is based on the Isabelle/jEdit. The upper part of the window shows parts of our OCL example and the lower part shows, in *shallow mode*, a *certified* "down-cast"-property of the model. Fig. 4 shows a human readable certificate that is generated, for the same property, in *deep mode* (the whole content is about 7 kLoC). This targets, e. g., developers improving the certificate generation while inspecting Fig. 5 line by line.

A noticeable characteristic of Isabelle's IDE layer (**?**) is the "continuous build and continuous check" functionality, where "check" means "certification" in our context. It enables the user to infer, e. g., the type information by just hovering over sub-expressions in an already checked area, or to infer in an OCL expression to jump where an operation is defined in the class model by clicking on it.

We provide several packages supporting various modeling tasks arising from the specific needs of UML/OCL seen as DSL:

– *Class Model Package* for declaring a UML data model, i. e., classes, associations, aggregations, enumerations.
– *Invariant & Operation Package* for declaring, in the context of an already defined class model, OCL class invariants

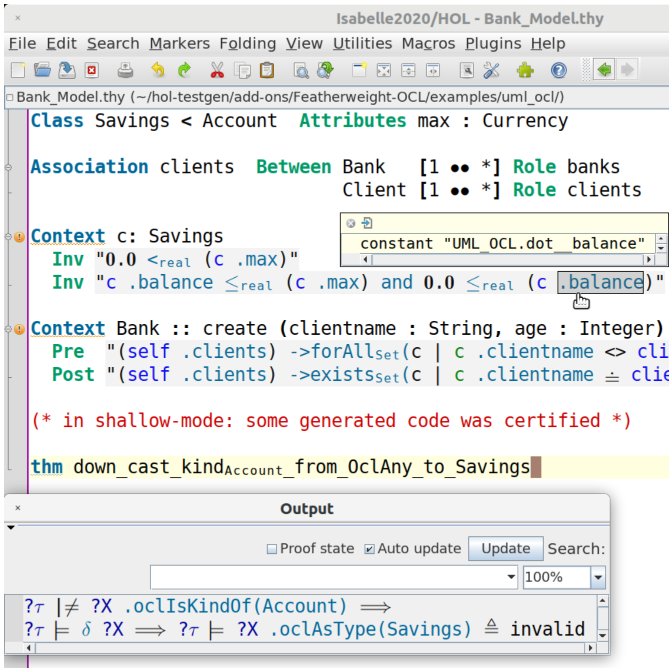


Fig. 3 User interface and shallow mode certification.

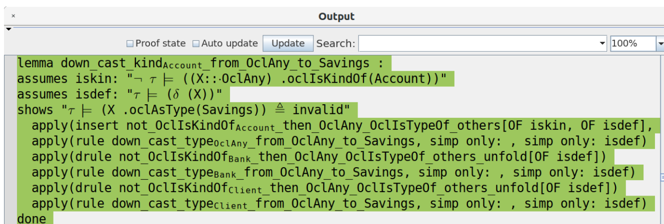
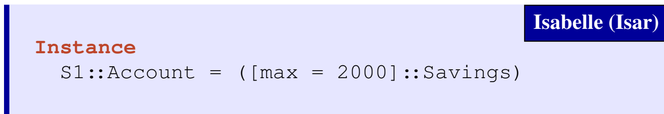


Fig. 4 Human readable certificate in deep mode (mostly targeting developers).

and operation contracts.

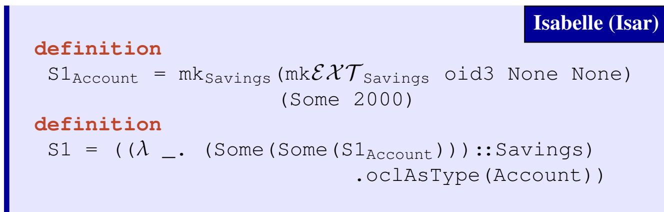
- *Instance Package* for declaring class instances (objects).
- *State Package* for grouping objects in a common state.
- *Transition Package* for transition properties over a pair of pre- and post-state.

After defining our data model using the Class Model Package (recall Fig. 3), we can use the Isar **Instance** command provided by the *Instance Package* for defining objects:



```
Isabelle (Isar)
Instance
  S1::Account = ([max = 2000]::Savings)
```

This command generates a set of definitions using the appropriate definitions in terms of our OCL library:



```
Isabelle (Isar)
definition
  S1Account = mkSavings (mkℰ𝒳𝒯Savings oid3 None None)
                          (Some 2000)
definition
  S1 = ((λ _. (Some(Some(S1Account)))::Savings)
                 .oclAsType(Account))
```

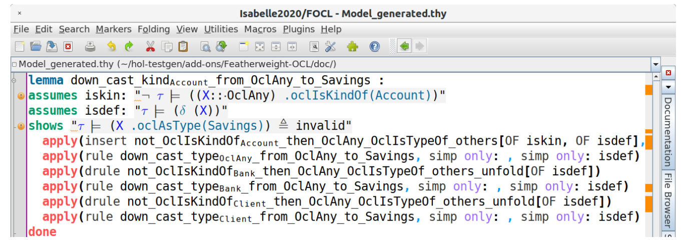
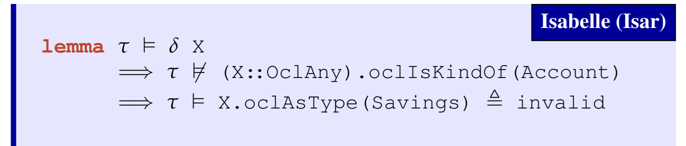


Fig. 5 Saving Fig. 4 and manually loading it in Isabelle/jEdit: it becomes now certified.

Besides definitions, packages generally also prove various user-defined properties (lemmas) over the UML/OCL model. In our example, the Class Model Package already proved that down-casting an object `X` from the topmost class **`OclAny`** to `Savings` does yield an error if `X` is not a subtype of `Account`:



```
Isabelle (Isar)
lemma τ ⊨ δ X
    ⟹ τ ⊭ (X::OclAny).oclIsKindOf(Account)
    ⟹ τ ⊨ X.oclAsType(Savings) ≜ invalid
```

Here, we write δ `X` for `not X.oclIsUndefined()`.

On a more pragmatic aspect, the provided packages were implemented with minimal efforts. For instance, in the following execution trace:

$$\text{Transition } [\bullet\bullet\bullet]\ \sigma_2$$

$$\rightsquigarrow$$

$$\text{State } \sigma_1 = [\bullet\bullet\bullet]$$
$$\text{Transition } \sigma_1\ \sigma_2$$

$$\rightsquigarrow$$

$$\text{Instance } X = \bullet\bullet\bullet$$
$$\text{State } \sigma_1 = [X]$$
$$\text{Transition } \sigma_1\ \sigma_2$$

where "• • •" represents a complex expression normally only understood by Instance. The particularity of our construction is that "• • •" becomes implicitly supported by State and Transition, without changing much of their implementations.

Fundamentally, our approach maintains the open and incremental character of Isabelle, particularly adapted to general users experimenting changing DSL constructs. However, beyond acting themselves as users, developers are also typically involved in extending user tools by new features, to implement different choices of "semantic deviation points" very common in the evolution of the UML, or to add generators for a new family of lemmas derivable from UML/OCL models. To this end, the meta-model construction described in Section 5 includes two modes specifically tailored for the needs of the two categories of users; they can be controlled by the generic command **`generation_syntax`** offering the two different options `shallow` and `deep`.

– In *shallow mode*, the system uses the Isabelle directly to encode a given UML/OCL model as well as to prove properties over this data model (certification). The proof prove strategies are derived from the model and the implementation makes use of Isabelle's code generator. Still, the trust in the system does not depend on the rather large code generator implementation: the certification only relies on the execution on methods of the small kernel of Isabelle, and, hence, only the kernel needs to be trusted. Also, as the kernel API is called without further overhead, this shallow mode is faster than deep mode. Consequently, shallow mode is the recommended for end-users of our system.
– In *deep mode*, for each UML/OCL command, an object-oriented datatype theory is generated (but not evaluated) using the concrete syntax of Isabelle/Isar. Usually, evaluating such large theory texts is significantly slower compared to using the internal API of Isabelle, which works on the abstract syntax of Isabelle/Isar. Hence, this mode is recommended for developers, who need to go through generated proof scripts in Isar on a step-wise basis, also to ensure that all generated specifications are the most faithfully representing their initial expectations. Fig. 4 (respectively Fig. 5) shows a fragment of such a generated proof.

Finally, for obvious consumer protection reasons, the delicate art of the exercise is to ensure that the loading of the generated object-oriented datatype theory implemented by developers (in deep mode) is really conforming to the sequence of functions later shipped and executed by end-users (in shallow mode).

To this end, we show how our original construction is drastically isolating the problem: it relies on implementing the two modes with a shared translation function T, making the difference in implementation of the two modes happening only after the execution of T. This approach makes any possible semantic variations between the two modes transparent on the level of the end-user DSL. To our knowledge, this technique supporting a DSL through a unified meta-model used to generate different presentations that are certified over different ways by the kernel has never been applied before in any existing proof assistant. This technique has never been applied to a language whose semantic interpretation is (deliberately) so versatile as is the case for UML.

4. Semantics of UML/OCL

Our tool is based on a library defining its core semantic concepts as a "shallow embedding" and a library of "built-in" OCL operations (**?**). In the following we will explain the core concepts such as the representation of OCL types in HOL to show how our generated definitions for states and objects interfere with them.

4.1. Core concepts

In general, a shallow embedding of object languages (such as OCL) into a meta-language (such as HOL) depends on the use of higher-order abstract syntax, i. e., binding in object-language terms is represented by λ-terms. Moreover, the type system of the object-language should be represented into types of the meta-language in an injective way, and the semantic representation of operators should respect this mapping. This way, only well-typed expressions of the object-language can be represented in the embedding, which liberates proof rules over the object language from side conditions referring to "well-typedness" of expressions, and which liberates the proof system from the need for type soundness (meta-)proofs. Overall, one can expect a far more efficient, application-oriented support for the object language, which is vital for proving or symbolic executions.

Types such as **Boolean**, **Integer**, and **Set**(X) are mapped to the corresponding (distinct) types `'𝔄 Boolean`, `'𝔄 Integer`, and $\texttt{Set}_{'\mathfrak{A}}\texttt{(X)}$. On the level of operations, for example, `not` or `and` having OCL types **Boolean** -> **Boolean** and (**Boolean**,**Boolean**) -> **Boolean** are defined as follows:

```
Isabelle (Isar)
definition
  not _ :: '𝔄 Boolean → '𝔄 Boolean
_ and _ :: '𝔄 Boolean → '𝔄 Boolean → '𝔄
    Boolean
```

The OMG standard (**?**) require all OCL types to possess explicit `invalid` and `null` elements (including the logic type **Boolean**). To represent this type requirement uniformly, we introduce the type classes `'α :: tc_null` to requiring the existence of two distinct elements `bottom` and `null`. Using the option type is a straightforward means to construct types in the class `tc_null` by "double lifting" basic types via `('α option) option`: any doubly lifted type is automatically an instance of the type class `tc_null`. This construction via type classes is flexible enough for constructions different from double-lifting, as is needed for sets, for example.

Since any OCL expression of type `T` may contain accessors to objects living in a pre-state and a post-state, OCL operations represent *functions* depending on a state pair yielding the corresponding type τ_{T} in HOL. This motivates the type synonym:

```
Isabelle (Isar)
type_synonym
  V'𝔄(τT) = '𝔄 state → '𝔄 state → τT::tc_null
```

We use this type synonym for constructing the types for OCL expressions (we discuss the precise form of `'𝔄 state` built over any object universe `'𝔄` in the next section). The type `'𝔄 Boolean` is an abbreviation for $\texttt{V}_{'\mathfrak{A}}$`((bool option) option)`, similarly `'𝔄 Integer` is an abbreviation for $\texttt{V}_{'\mathfrak{A}}$`((int option) option)`.

For example, the OCL logical constant `true` can now be defined as a function mapping any state transition pair τ to the (doubly lifted) truth value of HOL:

```
Isabelle (Isar)
definition
  true::V'𝔄('𝔄 Boolean) = λ τ. Some (Some True)
```

Analogously, we define the polymorphic constant `invalid` for all types of class `tc_null` by the constant function yielding `bottom`:

```
Isabelle (Isar)
definition
  invalid::V'𝔄('α::tc_null) = λ τ. bottom
```

Finally, we introduce a *validity* notion which is key for semantic invariants, pre- and postconditions. An expression E of type $'\mathfrak{A}$ *Boolean* is valid for a state-pair $\tau = (\sigma, \sigma')$ iff it evaluates to `true`:

```
Isabelle (Isar)
definition
  τ ⊨ E = (E (τ) =  true τ)
```

This is another way of saying that the transition τ is admitted by E. Based on validity, the semantics of an invariant can be given. For example, consider the following invariant (cf. Fig. 2):

```
OCL
Context c:Account Inv "0 < c.balance"
```

We represent this invariant in Isabelle/HOL by

```
Isabelle (Isar)
definition inv_Account(c)(τ) =
  (τ ⊨ 0 < c.balance ∧ τ ⊨ 0 < c.balance@pre)
```

Here `c` stands for any object of the class `Account` in the concrete object universe $\mathfrak{A}_{\texttt{Bank}}$ generated from the given `Bank` class. The operation `_ .balance` is induced from the class definition and has type $\texttt{V}_{\mathfrak{A}_{\texttt{Bank}}}(\texttt{Account}) \rightarrow \mathfrak{A}_{\texttt{Bank}}$ `Integer`; its type injectively represents the corresponding OCL type allowing type safeness for navigation on objects. The conjunction reflects the fact that the constraint must hold in both states. Since the arithmetic operation `_ < _` is strict with respect to `invalid` (as most OCL operators), it implicitly follows that `c.balance` must be defined and represent valid access to memory in both states; the main justification for the design decision to have a multi-valued logic for OCL is this particularly compact possibility to specify the valid parts in object graphs.

4.2. States and object universes

While generic library operations are defined to work on *all* object universes $'\mathfrak{A}$ represented by a polymorphic type variable, our *class model package* generates a concrete object universe $\mathfrak{A}_{\texttt{Bank}}$, instantiating the necessary generic operations (from the concrete syntax used in Fig. 2 and Fig. 3). In this object universe, semantic definitions can be given for the implicit operations mentioned in Section 4. The following elements are generated:

- type definitions for each class, which is a type of object instances comprising an object-identity; classes are organized in a non-reflexive partial inheritance relation (e. g., `Current < Account`),
- accessors for each attribute of a class, dereferenced in the pre-state and in the post-state (e. g., `_ .age@pre` and `_ .age`),
- accessors for each role-end of an association or aggregation (e. g., `_ .owner@pre` and `_ .owner`,
- cast operations `_ .oclAsType(C)` for each class `C` along the inheritance relation `_ < _`,
- test operations `_ .oclIsTypeOf(C)` for the *actual type* of an object, i. e., the type under which it is dynamically created. As in Java, this type does not change under casting.
- accessors `_ .allInstances()` returning the set of all object instances existing at some time in the state of a system,
- for each class `C`, there is a test `_ .oclIsKindOf(C)` which tests if the *dynamic type* of the given object belongs to one subtype of `C`,
- and finally a number of lemmas and proofs setting up the object-oriented datatype theory (?).

These definitions, lemmas and proofs refer to a typed denotational model, the *object universe*. In the work presented in this paper, we use a "closed world" version of the universe described in (??). Thus, our class model package provides, for a given class model, a concrete type instance for the generic object-universe $'\mathfrak{A}$ over which a state of objects (subsequently described) and all OCL operations are polymorphically parameterized.

The pivotal concept of *state* can now be defined as a pair consisting of a mapping from object identifiers *oid* to our object universe $'\mathfrak{A}$, i. e., the type comprising all formats of object instances, and a map of associations:

```
Isabelle (Isar)
record ('𝔄) state =
      heap :: oid ⇀ '𝔄
    assocs :: oid ⇀ oid list list list
```

Associations are potentially n-ary relations on objects; the map of associations encodes this by sorted lists of `oid`s.

The object universe is basically constructed as a sum type of the possible object instances for a given class diagram; the main technical difficulty of the construction is that up-and-down casts must be lossless in object-oriented datatype theories as discussed in the next section. This is realized by a suitable encoding of object extensions in each object instance possessing sub-classes; see (??) for details of this construction. This extensible approach, i.e., following (?) has also been used successfully for modeling large and widely-used data structures such as the Document Object Model (DOM) used in web browsers (?).

4.3. Properties of object-oriented datatype theories

As property of the object universe construction we obtain the following rule schema for all $\texttt{C}_i <^* \texttt{C}_j$ (i. e., the reflexive transitive closure of the subclass relation):

$$((X :: C_i).\texttt{oclAsType}(C_j).\texttt{oclAsType}(C_i)) = X$$

Meaning that whenever we cast up an object instance and cast it down again, we get the identity, i. e., casting is lossless. An instance of this scheme is:

Isabelle (Isar)

```
lemma
((X::Savings).oclAsType(Account)
                        .oclAsType(Savings))=X
```

where `X` is a free variable of the static (HOL) type `Savings`. The dual "down-up-cast" property is true under the precondition that the dynamic type test $X.\mathtt{oclAsType}(C_i)$ yields true.

Definitions of tests and casts are strict and neutral or idempotent on `null`. For example, our tool proves the instances of the lemma scheme:

$$(\mathtt{invalid} :: C_i).\mathtt{oclIsTypeOf}(C_j) = \mathtt{invalid}$$
$$(\mathtt{null} :: C_i).\mathtt{oclAsType}(C_j) = \mathtt{null}$$
$$(\mathtt{invalid} :: C_i).\mathtt{oclAsType}(C_j) = \mathtt{invalid}$$
$$(\mathtt{null} :: C_i).\mathtt{oclIsTypeOf}(C_j) = \mathtt{true}$$

Besides the lemmas on strictness and `null`-preservation, the relative position of C_i and C_j (in $C_i.\mathtt{oclIsTypeOf}(C_j)$) reveals opposite consequences (where we will write $\delta\ X$ for `not` $X.\mathtt{oclIsUndefined}()$ and, moreover, we will write $\upsilon\ X$ for `not` $X.\mathtt{oclIsInvalid}()$):

1. The type testing from a class C_i to a larger class C_j is always `false`. More precisely, for all classes $C_i <^+ C_j$ or $C_i \not<^* C_j$ and $C_i \not>^* C_j$:

$$\tau \models \delta\ X \Longrightarrow \tau \models ((X :: C_i).\mathtt{oclIsTypeOf}(C_j)) \triangleq \mathtt{false}$$

2. When reversing the inheritance relation between C_i and C_j, as soon as an object of the large class C_i *does* belong to the type of a small class C_j, the casting to C_j fails for all its sub-classes. For all $C_k <^+ C_j <^* C_i$ (or whenever $C_i \not<^* C_j$ and $C_i \not>^* C_j$):

$$\tau \models \delta\ X \Longrightarrow \tau \models (X :: C_i).\mathtt{oclIsTypeOf}(C_j) \Longrightarrow \tau \not\models \upsilon\,(X.\mathtt{oclAsType}(C_k))$$

5. Certified model transformation

As mentioned earlier, implementing support for specification constructs is usually done in the implementation layers of modern interactive theorem prover environments, so in SML for Isabelle/HOL (?) or HOL4 (?), OCaml for Coq (?), or Lisp for PVS (?). Despite notable improvements in programming technology, this did not change very much since the advent of the LCF-style provers in the 1970ies. However, the approach raises for some time concern with respect to its maintainability. There is a growing interest in meta-theoretic properties of implementations based on tactics such as termination and completeness. For example, it was observed that LCF-style component programming *"... requires intimate knowledge of the internals of the underlying theorem prover ... "* and moreover that *"... there is no way to check at compile time if the proofs will really compose ... "* (**?**).

Our model-based approach provides at least a partial answer to this problem, while remaining a pragmatic compromise: we use a *partial* model of the Isabelle API with a *partially* modeled semantics. We model the abstract syntax trees (ASTs) of terms, specification construction operations and tactics (proof construction), which allows us to formally describe the compilation functions and many well-formedness constraints. In contrast, the semantics of type inference and the tactic operations on formulas were *not* part of our Isabelle API model; these operations were represented as uninterpreted symbols. This does not exclude that a certain number of properties can be proved formally over the translation function. For example, the proofs over translation functions permit to formally establish their termination or compilation completeness results of the form "for well-formed input, the specification construct implementation will always produce a syntactic sequence of elementary theory extensions," which remains to be validated by the Isabelle kernel.

It is helpful to understand how an interactive prover such as Isabelle processes specification constructs: a *command* interpretation can be seen as a transaction of the underlying logical context σ, which contains among many other things the signature as well as all established theorems of a theory. Such a transaction is then a partial function of type $\sigma \rightharpoonup \sigma$; if the transaction fails (due to, e. g., type checking), particular Isabelle-specific error recovery techniques will try to cope with the partially incorrect theory, and try to find recovery entry points in case of modifications of the system input by the user.

The translation process covering the different paths and options is presented in Section 3. Our methodology to construct DSL specification constructs consists of the following steps:

1. We define the abstract syntax (AST) of its DSL as a collection of inductive datatypes in HOL; in UML and Model-driven Architecture (MDA) communities, this is called a "meta-model" ($\mathcal{M}_{\mathrm{DSL}}$) of the DSL.
2. We define a meta-model ($\mathcal{M}_{\mathrm{IsaAPI}}$) of the Isabelle kernel API tailored to our needs, and also conceived to overcome the aforementioned portability problem.
3. We define by a collection of function definitions the conversion $\mathcal{C}$ mapping $\mathcal{M}_{\mathrm{DSL}}$ to $\mathcal{M}_{\mathrm{IsaAPI}}$. Since $\mathcal{C}$ is defined in HOL, it can be made subject to a formal analysis by proofs in Isabelle/HOL itself.
4. We provide several setups for Isabelle's powerful code generator, most notably:
   - *Shallow mode*: This is SML code that is directly linked to the internal Isabelle API.
   - *Deep mode*: This is basically a textual pretty-print in Isabelle/Isar syntax.

For our concrete instance of the meta-model $\mathcal{M}_{\mathrm{DSL}}$, this comprises definitions for classes with invariants, associations, aggregations and operation constructs. Additionally, constructs for defining object models (concrete states) are made available.

The conversion $\mathcal{C}$ comprises both the generation of declarations, definitions in terms of denotational constructions, automated proofs for side conditions of type definitions and the generation of tactic support over the resulting theory. Since $\mathcal{C}$ produces a sequence of $\mathcal{M}_{\text{IsaAPI}}$ model elements that represent a composition of more elementary conservative specification constructs and proofs, the entire sequence will represent a conservative theory extension. This sequence can be seen as a *certificate* of the theory extension, which still has to be validated by the Isabelle kernel. Since invariants and operation contracts contain OCL library operations, the certification process—which includes type and proof checking—will be executed into a logical context containing the OCL library described in Section 4.

For the validation of the $\mathcal{M}_{\text{IsaAPI}}$-sequence, we proceed in two different ways: in the deep mode, the generated textual pretty-print of a theory file is imported by Isabelle, whereas in shallow mode the generated SML code is directly compiled by SML compiler and linked to the internal Isabelle API's. As it can be expected, the latter process has a substantially better performance and can be done behind the scenes in Isabelle. The resulting user experience with respect to IDE support and reactivity of the generated theory extension is sufficient for medium-sized models.

5.1. A meta-model for UML/OCL

Following our strategy to represent the models as abstract HOL entities, our meta-model $\mathcal{M}_{\text{UML}}$ is defined via a number of inductive data-type definitions in Isabelle/HOL. Here is a general overview with the main concepts, i. e., the data model

Isabelle (Isar)

```
datatype uml_type  =
  OclAny | String |  ... | Sequence uml_type |
    ...

datatype uml_class =
  Class  super_class: uml_class
         class_name : string
         attributes : (string * uml_type) list
| OclAny
```

as well as the behavioral aspects, i. e., the contracts:

Isabelle (Isar)

```
datatype uml_opn_contract =
 Contract context_class :string
          opn_sign :string*(string*uml_type)
    list
          preconds :(string*string) list
          postconds:(string*string) list
```

As a consequence, a part of our example (Fig. 2) has the abstract syntax term:

OCL

```
Class OclAny ''Bank''  [(''name'',String)]
```

To complete $\mathcal{M}_{\text{UML}}$ with the definition of invariants, associations and aggregations with their multiplicities we proceed analogously. Terms for preconditions and postconditions are still represented as strings, i. e., unvalidated syntactic representations of OCL formulas.

5.2. A meta-model for the Isabelle/HOL API

The $\mathcal{M}_{\text{IsaAPI}}$ meta-model supports the representation of

– types and terms (with syntax-declaration elements),
– elements for tactics and Isar high-level proof methods, and
– a selection of Isabelle commands (e. g., **datatype**, **lemma**, ...).

To give the reader an introduction to this API, consider Curry-style typed λ-terms:

$$T \;::=\; T\,T \mid \lambda\, V :: \tau.\, T \mid V :: \tau \mid C :: \tau$$

where τ are a set of type expressions, V is a set of free variable symbols and C is a set of constant symbols. The core of the Isabelle API's for λ-terms, which serve as a universal representation device for all formulas in Isabelle's logics, is a nearly one-to-one representation of this concept in SML:[1]

SML

```
datatype term = Appl of term * term
              | Abs of string * typ * term
              | Bound of int
              | Free of string * typ
              | Const of string * typ
```

Free variables are represented by the constructor `Free` in the λ-terms and Isabelle uses de Bruijn indices for bound variables:

$\lambda x :: \alpha.\, x$ is represented by `Abs("x",` typ_α`, Bound(0))`

The representation of the Isabelle API inside our HOL meta-model is not completely one-to-one, but still straightforward:

Isabelle (Isar)

```
datatype
 term = apply term (term list)
   | lambind string term
   | basic string
   | type_annotation term typ
   | if_then_else cond:term b1:term b2:term
   | fun_case (term option)((term × term) list)
   | let eqns:((term × term) list) body:term
   | term_context (string list) term
```

The constructor `basic` unifies constants and variables and suggests an untyped version; this facilitates practical term construction, but implies the call of Isabelle's type inference whenever a HOL model term is converted into an Isabelle API term to get inserted into the system. This is an additional logical safety-check for terms constructed from our meta-model. The constructor `type_annotation` stands for a term constrained by a type written in conceptual notation $t :: \tau$ which is a shortcut for $(\lambda x : \tau \to \tau.\ x)(t)$, while the other constructors are common shortcuts for larger HOL terms. For example, `if_then_else` *cond s t* is represented in the Isabelle API by `Appl(Const("If",_), Appl(`$term_{cond}$`,` `Appl(`$term_s$`,` $term_t$`)))`. The `term_context` plays a particular role: it simulates a pre-initialized context (de

[1] This presentation is slightly simplified; the real Isabelle λ-terms use additionally an infix-notation for applications and another constructor for a class of variable symbols relevant for deduction.

Bruijn variables under "lambda") and works as an explicit substitution (?); it is used to convert explicit free variables in de Bruijn indexes.

The constructors of proofs elements and theories (i. e., *global contexts*) are defined analogously:

Isabelle (Isar)
```
datatype tactic = rule  ... | erule ...
                | simp  ... | simp_only ...

datatype command = datatype ...
    | definition string * typ option * string
    | lemma name:sting goal:string
              proof:(tactic list)
    | consts name:string decl:typ
    | ...

type_synonym hol_theory =
     theory name:string parents:(string list)
              body:(command list)
```

This interface model $\mathcal{M}_{\text{IsaAPI}}$ is an abstraction of a functional "structure-signature API" used by Isabelle. As the latter builds the basis for many different variants of commands. The reader interested in the details to build a more complete set of supported commands is referred to (?).

5.3. Meta-model transformation

The definition of the conversion function is a more or less straightforward implementation of the coding scheme presented in Section 4. This conversion is not necessarily final, and might be adapted depending on the degree of automation initially expected by end users, e. g., kind of lemmas initially provided for a typical class model. To give an impression of this style of modeling (programming), we present a fragment of this compilation function: the part that generates the proof for an instance of the generic theorem schema:

$$\texttt{not}\ (X :: \texttt{OclAny}).\texttt{OclIsKindOf}(Y)\ \texttt{and}$$
$$\delta(X)\ \texttt{implies}\ \delta(Y.\texttt{OclIsTypeOf}(X))$$

for all $X, Y \in {}'\mathfrak{A}$. Recall that this theorem (which depends on the structure of the class model $'\mathfrak{A}$) rules out any combinations of invalid castings for any polymorphic universe.

The following function generates a proof certificate for this theorem in form of a tactic term and reads as follows:

Isabelle (Isar)
```
definition
  cons l tac = (if l=[] then (_#_) tac else id)

fun aux_depth and aux_breadth where
  aux_depth l_depth =
      (λ [] ⇒  []
       | (class, l_depth)#l_depth ⇒ simp_only class
                    # aux_breadth class [] l_depth (rev l
    _breadth)) l_depth
| aux_breadth class tactic l_depth l_breadth =
      (λ [] ⇒  tactic
    | (class0, class0_path)#l_breadth ⇒
                   erule (class, map fst l_depth) #
    cons l_breadth simp
                   ...
```

The rest of the construction for $\mathcal{C}$ proceeds similarly as shown above in HOL. It turns out the complete $\mathcal{C}$ function can be formally analyzed inside Isabelle/HOL itself; for example, we proved for all these functions their terminations, as well as the well-formed completeness of all class models used.

5.4. Subsequent certification techniques

From the conversion function $\mathcal{C}$ mapping $\mathcal{M}_{\text{DSL}}$ to $\mathcal{M}_{\text{IsaAPI}}$, we obtain *two* results corresponding to the shallow or deep mode. They are complementary from a certain perspective: once a value in $\mathcal{M}_{\text{IsaAPI}}$ has been computed, that value can be either immediately executed (bottom of Fig. 6), or converted to concrete Isar syntax (top of Fig. 6). It is a particular advantage of our approach that other code generator setups can be easily added, for example for OCaml, Haskell, or Scala (since these languages are supported by the code generator.) This may be of particular interest if external UML/OCL tools are already available for these languages.

5.4.1. Shallow mode.

The principle of compiling a formula with computational content to code (via code generation), evaluating it, and re-introducing the result in derivations over the formula is called *reflection*. We use reflection not only to produce parts of proofs, but to construct (parts of) the entire proof environment for UML/OCL inside Isabelle/HOL.

There is meanwhile a large body of publications on reflection, e. g., (?) for a general survey, and (?) for a universal axiomatizing approach. In Isabelle, one way to reflect HOL functions is by using the special command `code_reflect`. Isabelle's reflection mechanism is based on a very versatile code generator (??). On the basis of the SML environment extended by this generated code, we provide an integration of the conversions into the Isabelle/Isar top level to support the syntax shown in Fig. 6 which makes the resulting tool accessible to Isabelle users.

5.4.2. Deep mode.

The deep mode resembles to shallow mode, except it uses a different back-end for the validation of the certificate: in deep mode, the generated model is a pretty-print of an Isabelle/HOL theory which contains the resulting definitions of the object-oriented data model and the proofs for all derived theorems according to the lemma schemes in Section 4. The deep mode serves two purposes: first, it proves at times as a valuable tool for debugging, and second, it is a human readable certificate of the overall theory construction that may play a role in high-level certification processes, for example, as in CC EAL 6+ (?).

5.5. DSL implementations as model transformations

By using Isabelle/HOL as an "implementation language" for the model translator, one immediately benefits from verified algorithms for efficient data-structures (e. g., formalized red-black trees, or algorithms for computing transitive closures). The developer also profits of the possibility to *prove* properties over the generated compilation, albeit this is limited to termination and completeness proofs *modulo* validation. This still allows proofs of properties over syntactic and static sanity of the generated

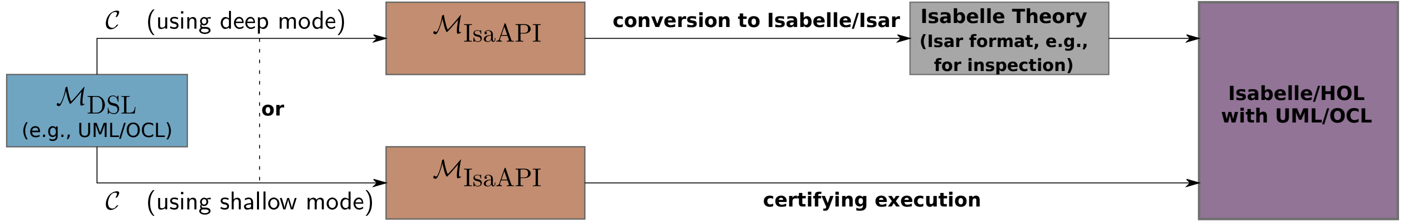


Fig. 6 The implementation of deep in the exportation scenario (top) and shallow in the reflection scenario (bottom).

functions and models (such as: "if no context error in the class package syntax occurred, it can be assured that all generated names for accessors were distinct" or slightly more challenging: "if the class model is well-formed, the generated code will be well-typed with respect to HOL types."). In case that future projects provide a complete model of the type checker and the core-inference engine of Isabelle, it is even conceivable to extend our approach by completeness proofs *including* validation assuring that the certification will not fail. For other systems in the LCF-prover family such as HOL4, such model of the deductive kernel are meanwhile available (**?**), albeit not (yet) for the Isabelle kernel.

6. Applications scenarios of UML/OCL

In this section, we will show how to apply the generated verified environment UML/OCL. In theory, UML/OCL is relatively complete to HOL, i. e., every formula ϕ that is valid in all Henkin-style models can also be proved `True` in UML/OCL, and the same restrictions to completeness with respect to standard models hold for OCL as for HOL; the reader interested in the foundations of HOL is referred to (**?**). From a practical point of view, UML/OCL is not equally suited to any possible application: additional automated proof support will be necessary, for example, in areas such as refinement proofs over different class models as discussed in (**?**). The main reason for this is that our semantics of class models uses object ids, which makes a "reasoning modulo graph isomorphism" necessary. Without substantial further support, such proofs tend to get complicated and are nothing for the faint-hearted. Moreover, simplistic proof strategy consisting in unfolding all definitions of OCL operators for a given class model leads to large formulas in which automated reasoning is restricted to a very low level and therefore doomed to fail.

6.1. Certified code generation and basic animation

Isabelle/HOL contains a powerful code generator that targets SML, Haskell, OCaml, and Scala. Adequately configured, it yields fairly efficient code for these languages, a feature that has been used for the generation of large practical programs, e. g., (**?**).

Moreover, the code generator is used in efficient proof tactics for proving formulas by code evaluation, see (**?**) for details. We used this feature to check ground formulas in OCL and in concrete data models. We implemented an **Assert** command that checks if a given ground-formula can be reduced to `True` simply by evaluation; the latter practically excludes operators that involve underspecified choice (such as `X->any()`) or operation constructs for non-deterministically specified operations. Since many operation contracts contain a postcondition of the form `result = E` for some expression `E`, that appears relatively often.

With respect to library operations, we used this feature for a number of semantic test cases or semantic checks for OCL library operations. The following presents an example of checking corner cases of the OCL **Set** operations:

Isabelle (Isar)
```
Assert
  τ ⊢ (Set{1,2} ≜ Set{}->including_Set(2)
                        ->including_Set(1))
Assert
  τ ⊢ (Set{1,invalid,2} ≜ invalid)
```

Here, the formulas are not even ground: the variable τ is just never used in the evaluation because the properties are state-independent, i. e., universal properties of the OCL library.

We also provide commands for specifying object model, i. e., instances of a class model:

Isabelle (Isar)
```
Instance
      c1::Client =
      [client_id=101 , name=''Alice'', age=25]
  and c2::Client =
      [client_id=102 , name=''Bob'', age=17]
  and a1::Account = [id=2100110, balance=500]
  and a2::Account = [id=3100500, balance=20]
  and a3::Account = [id=5010101, balance=1000]
  and b1::Bank = [bank_name=''Banco Fiasco'']
```

Which allows us to define, e. g., an initial system state:

Isabelle (Isar)
```
State σ1 =
 [([c1, c2]::Client),
  ([a1 with_only balance=600,a2,a3]::Account),
  ([b1]::Bank)]
 [b1 associates_to a1,a2,a3 via manages,
  c1 associates_to a1,a2 via owner,
  c2 associates_to a3 via owner]
```

Together with the above mentioned **Assert**-command, we can now explore assert on the state σ_1 properties such as **Assert**`(`σ_1`,`σ_1`) ⊢ a1 >= 550` or, similarly, that the class invariants shown in Fig. 2 hold.

OCL
```
Context
  Bank::withdraw (c : Client, account_id : Integer, amount:Integer)
  Pre  def: (δ c) and (δ account_id) and (δ amount)
  Pre  1 ≤int amount
  Pre  (self .managed_accounts)
       ->existsSet(X | (X .owner) ≐ c and ((X .account_id) ≐ account_id) and (amount ≤int (X .balance)))
  Post let A′ = self .managed_accounts
               ->selectSet(X | (X .owner) ≐ c and ((X .account_id) ≐ account_id))->anySet();
           A   = self .managed_accounts@pre
               ->selectSet(X | (X .owner) ≐ c and ((X .account_id) ≐ account_id))->anySet()
       in  (A′ .balance) ≐ (A .balance −int  amount)
  Post frame: let A = self .managed_accounts
                       ->selectSet(X | X .owner ≐ c and (X .account_id ≐ account_id))->anySet()
               in  ((Set{A}  ->oclIsModifiedOnly())
```

Fig. 7 The `withdraw` operation.

6.2. Establishing relations between invariants

It is possible to formally prove the relations between invariants. If no structural changes of the underlying class model are involved, or deeper reasoning over quantifiers, proofs can be fairly easy rewrite proofs based on the derived rules of the OCL library. For example, from

OCL
```
τ ⊨ 25 <= c.owner.age
            implies c.overdraft = 250
```

we can prove properties like

OCL
```
τ ⊨ 25 <= c.owner.age
            implies c.overdraft > 200
```

conclusively for all transitions between valid states.

6.3. Symbolic execution of sequence diagrams

It is possible to enchain also operations and reason over state-sequence charts. Assume that we extend our model by withdraw and deposit operations. Fig. 7 shows the withdraw operation (we omit the dual deposit operation).

Borrowing the concept of the sequencing operator *bind* from the state-exception monad well-known in functional programming, we can express the diagram in Fig. 8 together with some initial constraints as a proof goal (see Fig. 9). The lemma describes that in any state σ_0, both the clients `c1` and `c2` exist and have the right account number for their accounts, and precondition of the `withdraw` operation is fulfilled. We now can apply derived symbolic execution rules in this Isar proof:

Isabelle (Isar)
```
  apply(subst get_balance_Symbex, cleanup)
  apply(subst get_balance_Symbex, cleanup)
  apply(subst withdraw_Symbex, cleanup)
  apply(subst deposit_Symbex, cleanup)
  apply(subst get_balance_Symbex, cleanup)
  apply(subst get_balance_Symbex, cleanup)
  apply(subst explicit_assertion, auto)
done
```

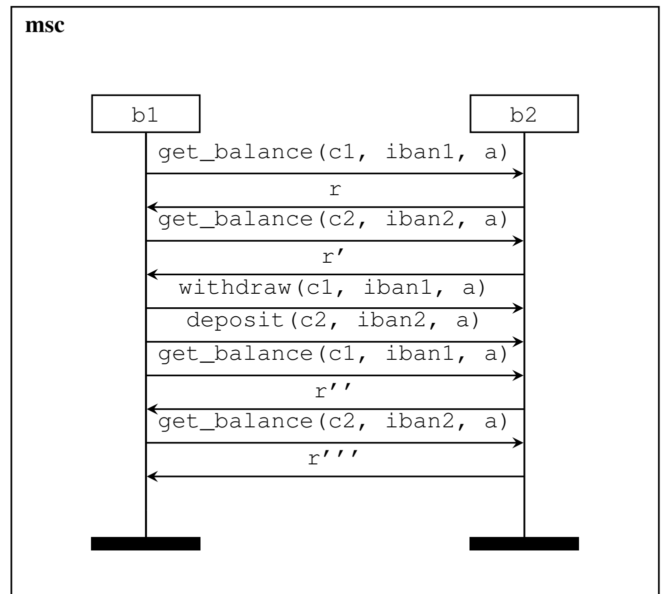


Fig. 8 Sequence diagram modeling an bank-transfer.

The proof executes the operations stepwise and accumulates the necessary constraints of these steps as assumptions. This proof not only shows that the final assertion "no money is lost in the transaction" is true, but also that the series of transactions is *valid*, i. e., each UML/OCL operation is possible in the state it is executed in, requiring that their preconditions are fulfilled.

7. Lessons learned

In this section, we discuss our experience and lessons learned in extending interactive theorem provers with support for DSLs. While our previous versions of HOL-OCL (**??**) are based on the datatype package approach, we use a reflection-based approach in the implementation presented in this paper. Thus, we can compare both approaches based on "first-hand" experiences.

```
Isabelle (Isar)
lemma valid_sequence2:
  assumes 1 :
        ∀ σ . (σ₀, σ) ⊢ bank .managed_accounts@pre
                       ->existsSet (X|X .owner@pre ≐ c1 and (X .account_id@pre ≐ a1))
  and   ∀ σ . (σ₀, σ) ⊢ bank .managed_accounts@pre
                       ->existsSet (X|X .owner@pre ≐ c2 and (X .account_id@pre ≐ a3))
  and   2: A  = bank .managed_accounts@pre
                    ->selectSet (X | (X .owner) ≐ c and ((X .account_id) ≐ account_id))->anySet ()
  and   3 : (σ₀, σ) ⊨ 0 ≤ A .balance
  shows
        σ₀ ⊢Mon  r    ::= bank .get_balance(c1 , a1) ;
                 r'   ::=  bank .get_balance(c2 , a3) ;
                 _    ::=  bank .withdraw(c1 , a1, a) ;
                 _    ::=  bank .deposit(c2, a3, a) ;
                 r''  ::=  bank .get_balance(c1 , a1) ;
                 r''' ::=  bank .get_balance(c2 , a3) ;
                 assertSE (λσ.  ((σ,σ) ⊢ (r +int r' ≐ r'' +int r''')))
```

Fig. 9 Proof goal representing the state chart in Fig. 8.

7.1. Certification and trust

We use HOL theories for generating *conservative extensions* of Isabelle/HOL, including new command-level Isar syntax allowing to access this functionality. From a user perspective, the result is very similar to a traditional datatype package, and we share the same trust guarantees: while both approaches cannot formally guarantee, e. g., the termination of "built-in" proof tactics or the completeness with respect to the DSL, the correctness only relies on the Isabelle kernel. If there are any logical errors in less trustworthy system parts (e. g., our Isabelle API binding, the Isabelle code generator), the kernel will reject the generated proofs. For example, let us assume that our reflection techniques generates, due to a bug in the code generator, a proof obligation that does not hold: the generated proof tactic will not be able to prove the statement, resulting in the top-level Isar command to fail, or to not terminate.

7.2. Development and maintenance time

The user interface of Isabelle changed a lot in the last two decades: from an interface that was mostly an SML-API used in a Read-Eval-Print-Loop (REPL) of the SML system to a system supporting a rich user interface and *structured proofs* that much more resemble the traditional mathematical notations (see (**?**) for a comparison of both proof styles).

While traditionally[2] datatype packages are conceptually very close to the apply-style approach, our reflection-based approach lifts the development of Isabelle extensions directly to the modern Isar level. Moving to the Isar level has several advantages that lower the barrier of developing packages as well as reduce the development and maintenance efforts:

- package developers do not need to learn a new interface of Isabelle (namely, the Isabelle's SML API).
- Isabelle's SML API is considered, by the Isabelle maintainers, as an "internal" API. Thus, documentation of changes is not as rich as for the Isar level. Changes of the Isar level are well documented and new Isabelle versions usually provide easy to follow step-by-step guides to adapt old theories to the latest version.
- if any generated proofs fail, the deep mode of our approach generates an actual Isabelle theory file that can be "debugged" interactively.

[2] This is changing in recent versions of Isabelle, which start to provide a SML API closer to the Isar level.

Table 1 Performance comparison: shallow vs. deep mode.

	#classes	#assocs	shallow [s]	deep [s]
Abstract List	1	0	47.02	53.70
Bank	6	3	119.22	136.90
Clocks Lib	6	0	138.86	158.23
Analysis	3	1	97.93	116.95
Design	3	1	98.32	113.83
Flight	5	3	126.50	149.12
Linked List	2	0	59.21	67.67

In contrast, the traditional datatype package requires a high level of expertise on the internal SML APIs.

7.3. Usability

The user experience follows the overall experience of Isabelle, including such details as type information when hovering over terms (recall Fig. 2). The advantages of our current work over previous versions of HOL-OCL are due to the fact our current work is based on the latest version of Isabelle: its usability improvements could also be implemented on top of a traditional datatype package, as used by HOL-OCL.

Still, it remains to show that the performance of our system is good enough. Table 1 shows a collection of UML/OCL examples of "typical" size in terms of classes and associations) (e. g., the examples shipped as part of USE (**?**) are of similar size). On a modern laptop using common hardware, the datatype package for the UML class models takes less than three minutes in gen-

eral for our examples. While this is rather long for interactive work, we do not consider this a show-stopper: during analyzing UML/OCL models, the UML data model part changes in our experience only rarely while the actual user-defined OCL need to be updated frequently. To support this work-flow, our system allows defining OCL constraints interactively and encoding a single OCL constraints usually takes less than a second. Overall, this allows for the interactive workflow that is typical for modeling and proving tasks in Isabelle.[3]

A closer look to the times (recall Table 1) reveals that deep mode is performing more operations than shallow. Indeed, in deep mode, generated Isabelle theories are presented using the concrete syntax of Isabelle: compared to shallow mode (where we work directly on the abstract syntax), deep requires an additional parsing step. On the other hand, since shallow requires itself an additional model transformation step, we conclude that the computation of the latter is taking less time than the parsing of the latter's serialized output.

While Isabelle is already utilizing parallelism features of modern multi-core processor architectures, most of them are actually only available in deep mode. In particular, *commands* being generated in shallow mode are still processed in Isabelle as atomic sequential actions. Future versions of Isabelle might overcome this limitation.[4]

8. Related work & conclusion

Related work. System components mapping specialized specification constructs to conservative extension schemes are known as part of LCF-architectures since the earliest HOL systems (**?**). In most systems, they offer support for, e. g., inductive datatypes, quotient types, and recursive function definitions. Our work distinguishes itself in two ways: firstly, we provide support for DSLs, and secondly by its implementation via meta-model transformation. Moreover, in contrast to a conventional packages targeting functional programming language support, we have to cope with the intricacies of object-orientation, involving sub-typing, object ids and referential equality, and the difficulties arising from states and state transitions.

Reflection as a concept to analyze proofs or proof generating functions (e. g., decision procedures for particular fragments of logics) using a second layer of logic, a meta-logic, has been known for a long time. Notable research ranges from (**?**) to work on self-formalization of HOL and corresponding prover implementations (**?**). Our work distinguishes itself from these approaches by modeling *partially* a real, sophisticated HOL system like Isabelle on the one hand and by restricting ourselves to fairly simple, but pragmatically important properties on the other (e. g., termination of the model transformation or totality of the compilation functions).

UML/OCL also attracts a lot of interest in various tools, mostly compilers such as Eclipse OCL (http://www.eclipse.org/modeling/mdt/ocl/). There are also some proof tools; however, they are mostly based on naive translations of the OCL syntax to the logic of a home-grown prover, disrespecting the OMG standard's many-valued logics as well as the semantics of library operations. An exception is (**?**), which combines a generation of axioms for the object-oriented datatype theory and SMT solving; we recall that our approach is strictly definitional. We presented, in this paper, a complete re-implementation of HOL-OCL (**?**). Our re-implementation supports a logically different, more recent version of OCL, utilizes the improvements of Isabelle's user interface, and is built using certified model transformations.

Take home message. We have shown a particular method to construct a series of theory support components (packages) by a particular technique rephrasing ideas from model transformation (in the sense of the MDA/MDE) into the more rigorous context of theorem proving. The technique has pragmatic as well as theoretic advantages: pragmatically, users of interactive theorem provers are becoming developers but can stay in the development framework they are used to, have better debugging facilities for generated tactic code, and can base their work on better abstractions of internal prover APIs that can be shared within the community. Theoretically, the approach does not introduce additional assumptions of trust since generated definitions and lemmas are finally certified by the original prover kernel. The approach lays the groundwork for proving properties over the translation itself: there is a full spectrum of possibilities ranging from termination and syntactic completeness proofs (e. g., the translation is a total function), down to semantic properties like "the evaluation of the generated proof certificates will not fail" or the correctness of the model transformation itself with respect to semantic interpretations of its input and output.

Availability. The formalized meta-model of the Isabelle API (**?**) and the resulting formalization of the OCL library (**?**), called Featherweight OCL, are both available as part of the Archive of Formal Proofs (https://www.isa-afp.org/) under a BSD license (SPDX-License-Identifier: BSD-3-Clause).

About the authors

Achim Brucker is a full Professor in Computer Science at the University of Exeter, UK and leading the Security and Trust of Advanced Systems research area. His research interests include formal methods, secure software engineering, and cyber security. He has 18 years of professional experience in these areas both in industry and academia. For six years, he was active in the OCL standardization process at the OMG. You can contact the author at a.brucker@exeter.ac.uk or visit https://www.brucker.ch/.

Frédéric Tuong is a Research Fellow at the Trinity College Dublin, Ireland. The heart of his research related to "UML/OCL model-theory-transformation in Isabelle" originated from a cooperation agreement between Université Paris-Saclay (replacing: Univ. Paris-Sud) and IRT SystemX. The resulting thesis of

[3] HOL-OCL (**?**) uses a more advanced encoding of UML data models using an open-world assumptions. This also supports the step-wise extension of the actual data model and could also be implemented in a reflection-based approach.

[4] We actually developed an experimental modification of Isabelle that shows a potential for significant performance improvements; see https://gitlab.scss.tcd.ie/tuongf/isabelle_para/.

this research (**?**) was therefore granted with funds of the Program "Investissements d'Avenir." You can contact the author at frederic.tuong@tcd.ie or visit https://www.scss.tcd.ie/frederic.tuong/.

Burkhart Wolff is a full Professor in Computer Science at the Université Paris-Saclay, France, and member of the VALS team Verification of Algorithms Languages and Systems. His interests are in formal modeling and verification techniques such as theorem proving and model-based testing. He implemented several models based on the Isabelle platform, both of the application and foundation side. You can contact the author at wolff@lri.fr or visit https://www.lri.fr/~wolff/.