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 (Paulson 1996) 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 (Bertot and Castéran 2004) or Isabelle (Nipkow et al. 2009) by user-friendly high-level languages such as Gallina (Coq 2019) for Coq or Isar (Wenzel 2002) 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 \((\Sigma, \Phi)\) of a signature \(\Sigma\) and a set of formulas \(\Phi\) (denoting axioms or theorems), a theory extension is a map \((\Sigma, \Phi) \mapsto (\Sigma \uplus \Sigma', \Phi \cup \Phi')\). A theory extension is conservative if 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 (Melham 1992)) is at the heart of this paper. We argue that in the context of semantics for domain-specific languages (DSLs),...
The notion of correctness implies obviously a lot of formal machinery which is in stark contrast to the simplicity and attractiveness 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 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 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 (Tuong and Wolff 2015) 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 (Tuong 2016), together with a functional working example in (Tuong and Wolff 2015).
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 (Paulson 1996) 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 t thm consists basically of triples of the form $\Gamma \vdash \theta \phi$, stating that the Boolean term $\phi$ could be derived in theory $\theta$ from the assumptions $\Gamma$, i.e., a list of Boolean terms. The t thm type is protected by the SML module system, which means that only kernel operations from this layer can produce t 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:

   \[
   \text{definition } c::\tau \text{ where } c = E
   \]

   Isabelle (Isar)

   Constant definitions can be seen as a kind of macro for the declaration of a constant symbol $c$ of type $\tau$, 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,
2. Type definitions, written as follows:

```isar
typedef '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 option` (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 (OMG 2011) and its textual extension OCL (OMG 2012) 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 (Gogolla et al. 2007).

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) (OMG 2012). OCL is a textual
language; its 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 (Wenzel 2002) and its IDE Isabelle/jEdit (Wenzel 2020).

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 (Wenzel 2014) 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.
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:

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

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

```
definition S1Account = mkSavings mkEXT Savings oid3 None None
```

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`:

```
lemma \( \tau \models \delta X \Rightarrow \tau \not\models (X::OclAny).oclIsKindOf(Account) \Rightarrow \tau \models X.oclAsType(Savings) \triangleq \text{invalid} \)
```

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

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

```
Transition [●●•] \( \sigma_2 \)
\( \Downarrow \)
State \( \sigma_1 = [●●•] \)
Transition \( \sigma_1 \sigma_2 \)
\( \Downarrow \)
Instance \( X = [●●•] \)
State \( \sigma_1 = [X] \)
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 variations between the two modes transparent on the level of

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 \( \lambda \)-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 Set\( _A (X) \). On the level of operations, for example, not and having OCL types Boolean \( \rightarrow \) Boolean and (Boolean, Boolean) \( \rightarrow \) Boolean are defined as follows:

\[
\text{definition}
\begin{array}{l}
\text{not} \_ \_ :: \ 'A \ Boolean \rightarrow \ 'A \ Boolean \\
\_ \_ \_ \_ :: \ 'A \ Boolean \rightarrow \ 'A \ Boolean \rightarrow \ 'A \ Boolean
\end{array}
\]

The OMG standard (OMG 2012) 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 '\( A \) :: 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 ('\( A \) 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 \( \tau \) in HOL. This motivates the type synonym:

\[
\text{type synonym}
\begin{array}{l}
V_\mathcal{A} (\tau_r) = 'A \ state \rightarrow 'A \ state \rightarrow \tau :: tc_null
\end{array}
\]

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 \( V_\mathcal{A} (\text{bool option}) \) option, similarly '\( A \) Integer is an abbreviation for \( V_\mathcal{A} (\text{int option}) \) option.

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

\[
\text{definition}
\begin{array}{l}
\text{true} :: V_\mathcal{A} (\text{'A Boolean}) = \lambda \tau. \text{Some (Some True)}
\end{array}
\]

Analogously, we define the polymorphic constant invalid for all types of class tc_null by the constant function yielding bottom:
While generic library operations are defined to work on Integer semantic definitions can be given for the implicit operations concrete syntax used in Fig. 2 and Fig. 3. In this object universe, our design decision to have a multi-valued logic for OCL is this access to memory in both states; the main justification for 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 \( \mathcal{U} \) represented by a polymorphic type variable, our class model package generates a concrete object universe \( \mathcal{U}_{\text{Bank}} \), 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 (Tuong 2016).

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 (Brucker and Wolff 2008a; Brucker 2007). Thus, our class model package provides, for a given class model, a concrete instance for the generic object-universe \( \mathcal{U} \) 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 \( \mathcal{U} \), i.e., the type comprising all formats of object instances, and a map of associations:

Associations are potentially \( n \)-ary relations on objects; the map of associations encodes this by sorted lists of olds.

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 (Brucker and Wolff 2008a; Longuet et al. 2014) for details of this construction. This extensible approach, i.e., following (Brucker and Wolff 2008a) has also been used successfully for modeling large and widely-used data structures such as the Document Object Model (DOM) used in web browsers (Brucker and Herzberg 2020).

4.3. Properties of object-oriented datatype theories

As property of the object universe construction we obtain the following rule schema for all \( C_i \subset C_j \) (i.e., the reflexive transitive closure of the subclass relation):

\[
((X :: C_i).oclAsType(C_j).oclAsType(C_i)) = X
\]

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

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

1. $(\text{invalid} :: C_i).\text{oclIsTypeOf}(C_i) = \text{invalid}$
2. $(\text{null} :: C_i).\text{oclIsType}(C_i) = \text{null}$
3. $(\text{invalid} :: C_i).\text{oclIsType}(C_i) = \text{invalid}$
4. $(\text{null} :: C_i).\text{oclIsTypeOf}(C_i) = \text{true}$

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

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

$$\forall \sigma. \sigma \vdash \delta X \iff \sigma \vdash (\langle X :: C_i \rangle.\text{oclIsTypeOf}(C_j)) \sqsupset \text{false}$$

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

$$\forall \sigma. \sigma \vdash \delta X \iff \sigma \vdash (\langle X :: C_i \rangle.\text{oclIsTypeOf}(C_j)) \implies \sigma \not\vdash \nu (X.\text{oclAsType}(C_k))$$

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 (Nipkow et al. 2009) or HOL4 (Slind and Norrish 2008), OCaml for Coq (Bertot and Castéran 2004), or Lisp for PVS (Owre et al. 1996). 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 ...” (Chiaieb and Nipkow 2005).

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 $\sigma$, 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 $\sigma \rightarrow \sigma$; 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” ($M_{\text{DSL}}$) of the DSL.
2. We define a meta-model ($M_{\text{IsaAPI}}$) 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 $M_{\text{DSL}}$ to $M_{\text{IsaAPI}}$. 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 $M_{\text{DSL}}$, 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 $C$ 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 $C$ 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

\[
\text{datatype \ uml\_type} = \ \text{OclAny} \ | \ \text{String} \ | \ \ldots \ | \ \text{Sequence\ uml\_type}\ | \ \ldots
\]

\[
\text{datatype \ uml\_class} = \ \text{Class}\ super\_class: \ uml\_class \\
\text{class\_name} : \ \text{string} \ |
\text{attributes : (string * uml\_type) list} \\
| \ \text{OclAny}
\]

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

\[
\text{datatype \ uml\_opn\_contract} = \ \text{Contract\ context\_class \ :\ string} \\
\text{opn\_sign : string * (string * uml\_type)} \\
\text{preconds : (string * string) list} \\
\text{postconds : (string * string) list}
\]

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

\[
\text{Class \ OclAny \ "'Bank'" \ [("'name'", String)]}
\]

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., \texttt{datatype, lemma,...}).

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

\[
T ::= T \ | \ \lambda V :: \tau. T \ | \ V :: \tau \ | \ C :: \tau
\]

where $\tau$ 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 $\lambda$-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:\(^1\)

\[
\text{datatype } \text{term} = \text{Appl} \ \text{of} \ \text{term} \ * \ \text{term} \\
| \text{Abs} \ \text{of} \ \text{string} \ * \ \text{typ} \ * \ \text{term} \\
| \text{Bound} \ \text{of} \ \text{int} \\
| \text{Free} \ \text{of} \ \text{string} \ * \ \text{typ} \\
| \text{Const} \ \text{of} \ \text{string} \ * \ \text{typ}
\]

Free variables are represented by the constructor \texttt{Free} in the $\lambda$-terms and Isabelle uses de Bruijn indices for bound variables:

\[
\lambda x :: \alpha. x \text{ is represented by } \text{Abs("x", typ}\alpha, \text{Bound}(0))
\]

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

\[
\text{datatype } \text{term} = \text{apply} \ \text{term} \ \text{term} \ \text{list} \\
| \text{lambind} \ \text{string} \ \text{term} \\
| \text{basic} \ \text{string} \\
| \text{type\_annotation} \ \text{term} \ \text{typ} \\
| \text{if\_then\_else} \ \text{cond}\text{term} \ \text{b1}\text{term} \\
| \text{fun\_case} \ \text{term} \ \text{option} \ \text{term} \ \text{list} \\
| \text{let\ eqns}(\text{term} \ \text{list}) \ \text{body}\text{term} \\
| \text{term\ context} \ \text{string}\ text\ term
\]

The constructor \texttt{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 \texttt{type\_annotation} stands for a term constrained by a type written in conceptual notation $t :: \tau$ which is a shortcut for $(\lambda x :: \tau. t)(x)$, while the other constructors are common shortcuts for larger HOL terms. For example, if $\text{then\_else\ cond\ s\ t}$ is represented in the Isabelle API by $\text{Appl}(\text{Const("If",\_)}, \ \text{Appl}(\text{term}\_\text{cond}, \ \text{Appl}(\text{term}\_\text{then}, \ \text{term}\_\text{else})))$. The \texttt{term\ context} plays a particular role: it simulates a pre-initialized context (de

\(^1\)This presentation is slightly simplified; the real Isabelle $\lambda$-terms use additionally an infix-convention for applications and another constructor for a class of variable symbols relevant for deduction.
Brujin variables under “lambda”) and works as an explicit substitution (Abadi et al. 1991); it is used to convert explicit free variables in de Brujin indexes. The constructors of proofs elements and theories (i.e., global contexts) are defined analogously:

```
datatype tactic = rule ... | erule ... |
| simp ... | simp_only ...

datatype command = datatype ...
| definition string * typ option * string |
| lemma name:string 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 $M_{\text{IsaAPI}}$ 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 (Tuong and Wolff 2015).

### 5.3. Meta-model transformation

The definition of the conversion function is more or less a theorem in form of a tactic term and reads as follows:

```
the structure of the class model $A$ for all $\delta$ of the generic theorem schema:
modeling (programming), we present a fragment of this compi-
lation, albeit this is limited to termination and completeness proofs modulo validation. This still allows proofs

5.4. Subsequent certification techniques
From the conversion function $C$ mapping $M_{\text{DSL}}$ to $M_{\text{IsaAPI}}$, we observe two results corresponding to the shallow or deep mode. They are complementary from a certain perspective: once a value in $M_{\text{IsaAPI}}$ 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., (Costantini 2002) for a general survey, and (Clavel and Meseguer 1996) 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 (Chaieb and Nipkow 2005; Haftmann and Nipkow 2010). 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+ (Common Criteria 2006).

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 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 (Fallenstein and Kumar 2015), 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 \( \phi \) that is valid in all Henkin-style models can also be proved \( \text{True} \) 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 (Andrews 2002). 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 (Brucker and Wolff 2009). 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., (Brunner and Lammich 2018).

Moreover, the code generator is used in efficient proof tactics for proving formulas by code evaluation, see (Haftmann and Bulwahn 2020) 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 \( \text{True} \) simply by evaluation; the latter practically excludes operators that involve underspecified choice (such as \( X \rightarrow \text{any}() \)) or operation constructs for non-deterministically specified operations. Since many operation contracts contain a postcondition of the form \( \text{result} = E \) 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 \( \tau \vdash (\text{Set}(1,2) \triangleq \text{Set}() \rightarrow \text{including}_{\text{Set}}(2)) \rightarrow \text{including}_{\text{Set}}(1)) \)

Assert \( \tau \vdash (\text{Set}(1,\text{invalid},2) \triangleq \text{invalid}) \)
```

Here, the formulas are not even ground: the variable \( \tau \) 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

\text{c1}::\text{Client} =
\{\text{client_id}=101, \text{name}='\text{Alice}', \text{age}=25\}
\text{and} \text{c2}::\text{Client} =
\{\text{client_id}=102, \text{name}='\text{Bob}', \text{age}=25\}
\text{and} \text{a1}::\text{Account} = \{\text{id}=2100110, \text{balance}=500\}
\text{and} \text{a2}::\text{Account} = \{\text{id}=3100500, \text{balance}=20\}
\text{and} \text{a3}::\text{Account} = \{\text{id}=5010101, \text{balance}=1000\}
\text{and} \text{b1}::\text{Bank} = \{\text{bank_name}='\text{Banco Fiasco}'\}
```

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

```
Isabelle (Isar)

State \( \sigma_1 = \{\{\text{c1}, \text{c2}::\text{Client},
\{\text{a1 with only balance}=600,\text{a2},\text{a3}::\text{Account},
\{\text{b1}::\text{Bank})
\text{b1 associates to a1,a2,a3 via manages,}
\text{c1 associates to a1,a2 via owner,}
\text{c2 associates to a3 via owner}\}\}
```

Together with the above mentioned Assert-command, we can now explore assert on the state \( \sigma_1 \) properties such as \( \text{Assert} (\sigma_1, \sigma_1) \vdash \text{a1} \geq 550 \text{ or, similarly, that the class invariants shown in Fig. 2 hold.} \)
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

\[ \tau \models 25 \leq c\text{.owner\text{.age}} \implies c\text{.overdraft} = 250 \]

we can prove properties like

\[ \tau \models 25 \leq c\text{.owner\text{.age}} \implies c\text{.overdraft} > 200 \]

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 \texttt{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 \( \sigma_0 \), both the clients \( c_1 \) and \( c_2 \) 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:

\begin{Verbatim}
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
\end{Verbatim}

\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 (Brucker and Wolff 2008a,b) 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.
lemma valid_sequence2:
assumes 1 : ∀ σ . (σ₀, σ) ⊢ bank .managed_accounts@pre
→ exists_set (X | X .owner@pre = c1 and (X .account_id@pre = a1))
and 2: A = bank .managed_accounts@pre
→ exists_set (X | X .owner@pre = c2 and (X .account_id@pre = a3))
and 3 : (σ₀, σ) ⊨ 0 ≤ A .balance
shows σ₀ ⊨_mon r  := bank .get_balance(c1 , a1) ;
r'  := bank .get_balance(c2 , a3) ;
r'' := bank .withdraw(c1 , a1, a) ;
r''' := bank .deposit(c2, a3, a) ;
assert SE (λσ . ( (σ₀, σ) ⊨ (r + int r' = r'' + int r''')))
less than three minutes in general 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 workflow, our system allows defining OCL constraints interactively and encoding a single OCL constraint usually takes less than a second. Overall, this allows for the interactive workflow that is typical for modeling and proving tasks in Isabelle.

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.

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 (Gordon and Melham 1993). 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 (Harrison 1995) to work on self-formalization of HOL and corresponding prover implementations (Kumar et al. 2016). 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 (Arjona et al. 2014), 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 (Brucker and Wolff 2008a). 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 (Tuong and Wolff 2015) and the resulting formalization of the OCL library (Brucker et al. 2014), 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).

References
Abadi, M., Cardelli, L., Curien, P., and Lévy, J. (1991). Explicit substitutions. J. Funct. Program., 1(4):375–416.
Andrews, P. B. (2002). Introduction to Mathematical Logic and Type Theory: To Truth through Proof. Kluwer Academic Publishers.
Arjona, M., Dania, C., Egea, M., and Maña, A. (2014). Validation of a security metamodel for the development of cloud applications. In Brucker, A. D., Dania, C., Georg, G., and Gogolla, M., editors, Proceedings of the MODELS 2014 OCL Workshop (OCL 2014), volume 1285 of CEUR Workshop Proceedings. CEUR-WS.org.
Bertot, Y. and Castéran, P. (2004). *Interactive Theorem Proving and Program Development. Coq’Art: The Calculus of Inductive Constructions*. Springer.

Brucker, A. D. (2007). *An Interactive Proof Environment for Object-oriented Specifications*. PhD thesis, ETH Zurich. ETH Dissertation No. 17097.

Brucker, A. D. and Herzberg, M. (2020). A formally verified model of web components. In Jongmans, S.-S. and Arbab, F., editors, *Formal Aspects of Component Software (FACS)*, number 12018 in LNCS. Springer-Verlag.

Brucker, A. D., Tuong, F., and Wolff, B. (2014). Featherweight OCL: A proposal for a machine-checked formal semantics for OCL 2.5. *Archive of Formal Proofs*. http://www.isa-afp.org/entries/Featherweight_OCL.shtml, Formal proof development.

Brucker, A. D. and Wolff, B. (2008a). An extensible encoding of object-oriented data models in HOL. *Journal of Automated Reasoning*, 41:219–249.

Brucker, A. D. and Wolff, B. (2008b). HOL-OCL – A Formal Proof Environment for UML/OCL. In Fiadeiro, J. and Inverardi, P., editors, *Fundamental Approaches to Software Engineering (FASE08)*, number 4961 in LNCS, pages 97–100. Springer.

Brucker, A. D. and Wolff, B. (2009). Semantics, calculi, and analysis for object-oriented specifications. *Acta Informatica*, 46(4):255–284.

Brunner, J. and Lammich, P. (2018). Formal verification of an executable LTL model checker with partial order reduction. *J. Autom. Reasoning*, 60(1):3–21.

Chaieb, A. and Nipkow, T. (2005). Verifying and reflecting quantifier elimination for presburger arithmetic. In Sutcliffe, G. and Voronkov, A., editors, *Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)*, volume 3835 of LNCS, pages 367–380. Springer.

Clavel, M. G. and Meseguer, J. (1996). Axiomatizing reflective logics and languages. In *Reflection*, pages 263–288.

Common Criteria (2006). Common criteria for information technology security evaluation (version 3.1), Part 3: Security assurance components. Available as document CCMB-2006-09-003.

Coq (2019). The Coq proof assistant (version 8.10.0).

Costantini, S. (2002). Meta-reasoning: A survey. In Kakas, A. C. and Sadri, F., editors, *Computational Logic: Logic Programming and Beyond, Essays in Honour of Robert A. Kowalski*, Part II, volume 2408 of LNCS, pages 253–288. Springer.

Fallenstein, B. and Kumar, R. (2015). Proof-producing reflection for HOL - with an application to model polymorphism. In Urban, C. and Zhang, X., editors, *Interactive Theorem Proving (ITP)*, volume 9236 of LNCS, pages 170–186. Springer.

Gogolla, M., Büttner, F., and Richters, M. (2007). USE: A UML-based specification environment for validating UML and OCL. *Sci. Comput. Program.*, 69(1-3):27–34.

Gordon, M. J. C. and Melham, T. F. (1993). *Introduction to HOL: a theorem proving environment for higher order logic*. Cambridge University Press.

Haftmann, F. and Bulwahn, L. (2020). Code generation from Isabelle/HOL theories.

Haftmann, F. and Nipkow, T. (2010). Code generation via higher-order rewrite systems. In *Functional and Logic Programming (FLOPS)*, volume 6009 of LNCS, pages 103–117. Springer.

Harrison, J. (1995). Metatheory and reflection in theorem proving: A survey and critique. Technical Report CRC-053, SRI Cambridge.

Kumar, R., Arthan, R., Myreen, M. O., and Owens, S. (2016). Self-formalisation of higher-order logic. *Journal of Automated Reasoning*, 56(3):221–259.

Longuet, D., Tuong, F., and Wolff, B. (2014). Towards a tool for featherweight OCL: A case study on semantic reflection. In Brucker, A. D., Dania, C., Georg, G., and Gogolla, M., editors, *Proceedings of the MODELS 2014 OCL Workshop (OCL 2014)*, volume 1285 of CEUR Workshop Proceedings, pages 43–52. CEUR-WS.org.

Melham, T. F. (1992). A package for inductive relation definitions in HOL. In Archer, M., Joyce, J. J., Levitt, K. N., and Windley, P. J., editors, *International Workshop on the HOL Theorem Proving System and its Applications (TPHOLS)*, pages 350–357. IEEE Computer Society.

Nipkow, T., Paulson, L. C., and Wenzel, M. (2009). Isabelle’s logic: HOL.

OMG (2011). UML 2.4.1: Infrastructure specification. Available as OMG document formal/2011-08-05.

OMG (2012). UML 2.3.1 OCL specification. Available as OMG document formal/2012-01-01.

Owre, S., Rajan, S., Rushby, J. M., Shankar, N., and Srinivas, M. K. (1996). PVS: Combining specification, proof checking, and model checking. In Alur, R. and Henzinger, T. A., editors, *Computer Aided Verification (CAV)*, volume 1102 of LNCS, pages 411–414. Springer.

Paulson, L. C. (1996). *ML for the Working Programmer*. Cambridge Press.

Slind, K. and Norrish, M. (2008). A brief overview of HOL4. In Mohamed, O. A., Muñoz, C. A., and Tahar, S., editors, *Theorem Proving in Higher Order Logics (TPHOLs)*, volume 5170 of LNCS, pages 28–32. Springer.

Tuong, F. (2016). *Constructing Semantically Sound Object Logics for UML/OCL Based Domain-Specific Languages*. PhD thesis, University Paris-Sud. École Doctorale No. 580.

Tuong, F. and Wolff, B. (2015). A meta-model for the isabelle api. *Archive of Formal Proofs*.

van Heijenoort, J. (2002). *From Frege to Gödel : A Source Book in Mathematical Logic, 1879-1931 (Source Books in the History of the Sciences)*. Harvard University Press.

Wenzel, M. (2014). Asynchronous user interaction and tool integration in isabelle/PIDE. In Klein, G. and Gamboa, R., editors, *Interactive Theorem Proving (ITP)*, volume 8558 of LNCS, pages 515–530. Springer.

Wenzel, M. (2020). Isabelle/jEdit. Part of the Isabelle distribution.

Wenzel, M. M. (2002). *Isabelle/Isar — a versatile environment for human-readable formal proof documents*. PhD thesis, TU München.
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 this research (Tuong 2016) 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/.