Towards a Tool for Featherweight OCL: A Case Study On Semantic Reflection

Delphine Longuet[1], Frédéric Tuong[2] and Burkhart Wolff[1]

[1] Univ Paris-Sud, LRI UMR8623, Orsay, F-91405
CNRS, Orsay, F-91405
`{delphine.longuet, burkhart.wolff}@lri.fr`
[2] Univ Paris-Sud, IRT SystemX, 8 av. de la Vauve, Palaiseau, F-91120
`frederic.tuong@{u-psud, irt-systemx}.fr`

Abstract We show how modern proof environments comprising code generators and reflection facilities can be used for the effective construction of a *tool* for OCL. For this end, we define a UML/OCL meta-model in HOL, a meta-model for Isabelle/HOL in HOL, and a compiling function between them over the vocabulary of the libraries provided by Featherweight OCL. We use the code generator of Isabelle to generate executable code for the compiler, which is bound to a USE tool-like syntax integrated in Isabelle/Featherweight OCL. It generates for an arbitrary class model an object-oriented datatype theory and proves the relevant properties for casts, type-tests, constructors and selectors automatically.
Keywords: UML, OCL, Formal Semantics, Isabelle/HOL, Reflection.

1 Introduction

In this paper, we present a radically different approach to design, model, and implement a tool for UML and OCL. In the conventional approach, developers use hand-written Java programs complemented by the components generated by parsing generators [8] and by UML frameworks like OMG's MetaObject Facility (MOF)[10] or Eclipse Modeling Framework Project (EMF)[9]. In contrast, in our approach, we use the Isabelle Framework to generate a tool derived from a formal semantics for (some part of) UML/OCL based on earlier work for Featherweight OCL [3, 4, 6]. While the resulting tool — including user interface, document generator, code generator, and proof support — cannot compete to direct implementations such as [1, 12] with respect to speed of code-generation and completeness of the supported OCL language, our front-end has its value as a semantically founded reference implementation.

As a formalized theory in Isabelle/HOL, Featherweight OCL provides:

- the *algebraic layer* that contains the definitions of the four-valued logics (including `invalid` and `null`) with two equalities as well as theories for Integers and Sets,
- the *state layer* describing state-related operations like `allInstances()`, and
- the *object-oriented data-type layers* giving semantics to UML class models over this, comprising the theory of accessors, type casts and tests.

In this paper, we target to mechanize the latter. This means that our tool generates proven theorems that make the properties of object-oriented data-type theories explicit and which are needed as background-theory for other tools based on automated deduction striving for semantic compliance with the standard. As input, our tools takes a *class model*, which comprises:

1. Classes and class names (written as C_1, ..., C_n), which become types of data in OCL. Class names declare two projector functions to the set of all objects in a state: C_i`.allInstances()` and C_i`.allInstances@pre()`,
2. an inheritance relation $\_ < \_$ on classes and a collection of attributes A associated to classes,
3. two families of accessors for each attribute a and object X in a class definition (denoted by $X.a :: C_i \to A$ and $X.a$`@pre` $:: C_i \to A$ for $A \in \{\text{Boolean}, \text{Integer}, \ldots, C_1, \ldots, C_n\}$),
4. two families of operation declarations f_{op} for each class,
5. type casts that can change the static type of an object of a class (denoted by X`.oclAsType(`C_i`)` of type $C_j \to C_i$)
6. two dynamic type tests (denoted by X`.oclIsTypeOf(`C_i`)` and X`.oclIsKindOf(`C_i`)`),
7. and last but not least, for each class name C_i, an instance of the overloaded referential equality (written $\_ \doteq \_$).

We use the notation $e :: \tau$ to say that some expression e has the static type τ. Note that the phenomenon of dynamic types ("the type of an object at creation time") vs. static types ("the type inferred by the type inference in presence of possibly implicit casts") is characteristic for statically typed object-oriented languages such as Java or C++; apart from syntax, the object-oriented data model presented here is in no way specific to UML/OCL. Finally, for each function induced by the class model rules must be derived treating strictness, null, definedness, etc... Moreover, the subtype and inheritance relationship between objects must be expressed, for example by the rule:

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

where X is an object and C_k is a sub-type of C_j (i. e. $C_k < C_j$). This rule means that objects can be losslessly cast up and down again; this property is the key for the implementation of generic classes in Java and should also hold in UML.

While in [4] we described the *construction* of object-oriented data-types for UML class models *conceptually* and demonstrated it by an example containing definitions and rules proven by hand, we go in the present work one step further: We effectively*construct* an Isabelle plug-in for UML class models, that parses them in concrete input syntax (inspired by the USE tool) and compiles them to the necessary definitions, proofs, and infrastructure for execution and animation. As a by-product, UML class models can be directly edited *inside* Isabelle theory files, thus inheriting the Isabelle infrastructure including IDE, code generators, document generators and — last but not least — the proof environment for modeling and reasoning.

2 Background: Isabelle and UML/OCL

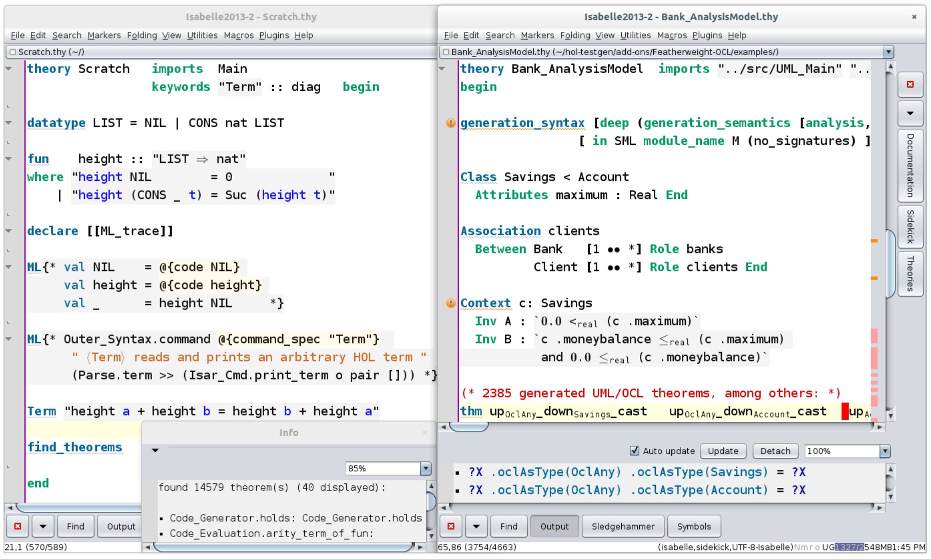
2.1 Isabelle: A Guided Tour through the Framework

In this paper, we want to emphasize the use of Isabelle as a generic **technical** framework. As such, it offers the possibility to "drive" the core-engine by user-programmed Standard ML (SML) programs in a logically safe way; a system configuration ("session") can thus contain logical definitions, proofs, text-documentation as well as code for tactic support as well as system extensions.

In order to demonstrate some relevant system features, we present a screenshot in Fig. 1. The left window shows a session based on Isabelle/HOL that consists of the only file `Scratch.thy`. One recognizes the header including theory and the "`imports` *Main*" clause ("*Main*" is a synonym for Isabelle/HOL) and then a sequence of subsections — called *commands* — introduced by a blue keyword: datatype, fun, declare, ML, find_theorems, thm... User-interaction to Isabelle is *document oriented*, i. e. each file belonging to a session is annotated by the prover while editing it as usual by modern IDEs. This annotation can consist, for example, in:

- colors (here: the underlying white indicates that Isabelle checked these commands and executed them without error),
- types (to be explored by tool-tips via the hovering-gesture),
- or values associated to computations inside these commands (displayed in the "Output" window when pointing to them; see Fig. 1 (right-below)).

Isabelle sessions can be extended by *user-defined commands*, a feature we use for defining Term, or for our own textual class model syntax in Fig. 1 (right):


Isabelle2013-2 - Scratch.thy
File Edit Search Markers Folding View Utilities Macros Plugins Help
Scratch.thy (~/)
Info
85%
Find
Output
21,1 (570/589)
Isabelle2013-2 - Bank_AnalysisModel.thy
File Edit Search Markers Folding View Utilities Macros Plugins Help
Bank_AnalysisModel.thy (~/hol-testgen/add-ons/Featherweight-OCL/examples/)
Documentation
Sidekick
Theories
Auto update
Update
Detach
100%
Find
Output
Sledgehammer
Symbols
65,86 (3754/4663)
(isabelle,sidekick,UTF-8-Isabelle)Nmro UG1327/2548MB1:45 PM

```
theory Scratch   imports  Main
                 keywords "Term" :: diag   begin

datatype LIST = NIL | CONS nat LIST

fun    height :: "LIST ⇒ nat"
where "height NIL         = 0             "
    | "height (CONS _ t) = Suc (height t)"

declare [[ML_trace]]

ML{* val NIL    = @{code NIL}
     val height = @{code height}
     val _      = height NIL     *}

ML{* Outer_Syntax.command @{command_spec "Term"}
       " ⟨Term⟩ reads and prints an arbitrary HOL term "
       (Parse.term >> (Isar_Cmd.print_term o pair [])) *}

Term "height a + height b = height b + height a"

find_theorems

end
```

```
found 14579 theorem(s) (40 displayed):

• Code_Generator.holds: Code_Generator.holds
• Code_Evaluation.arity_term_of_fun:
```

```
theory Bank_AnalysisModel  imports "../src/UML_Main" "..
begin

generation_syntax [deep (generation_semantics [analysis,
                   [ in SML module_name M (no_signatures) ]

Class Savings < Account
  Attributes maximum : Real End

Association clients
  Between Bank   [1 •• *] Role banks
          Client [1 •• *] Role clients End

Context c: Savings
  Inv A : `0.0 <_real (c .maximum)`
  Inv B : `c .moneybalance ≤_real (c .maximum)
           and 0.0 ≤_real (c .moneybalance)`

(* 2385 generated UML/OCL theorems, among others: *)
thm up_OclAny_down_Savings_cast   up_OclAny_down_Account_cast   up_A
```

```
• ?X .oclAsType(OclAny) .oclAsType(Savings) = ?X
• ?X .oclAsType(OclAny) .oclAsType(Account) = ?X
```

Fig. 1. Screenshot of Isabelle/jEdit: Standard HOL (left) and UML/OCL (right)

In the following, we describe the effect of commands in more detail (an in-depth treatment can be found in the implementation manual); the resulting class model compiler will be an application of the techniques demonstrated here.

The datatype command. The easiest way to understand this command is to view it as a kind of macro (albeit its syntax is inspired by functional programming languages) for the type declaration of `LIST` and a number of constant definitions and theorems. Some of these definitions construct a model of the constructors and derive its properties such as $\texttt{NIL} \neq \texttt{CONS}\ n\ mm$, $\texttt{CONS}\ n\ mm = \texttt{CONS}\ n'\ mm' \to n = n'$ and the induction rule for the type `LIST`. Thus, this command constructs the *theory* of a freely generated data-type.

The fun command. Similarly, the fun command allows for the declaration of recursive functions with pattern-matching. Again a conservative construction using a recursor is used; the derivation of the equations $\texttt{height NIL} = 0$ and $\texttt{height}(\texttt{CONS}\ n\ m) = \texttt{Suc}(\texttt{height}\ m)$ is done automatically involving a termination proof. This involved construction assures logical safeness: in general, just adding axioms for recursive equations causes inconsistency for non-terminating functions. The resulting equations can now be used in the Isabelle simplifier.

The ML command. Isabelle itself is built on top of an SML execution environment, accessible with the ML command: ML{*`3 + 4`*} compiles "`3 + 4`", executes it, and displays the result in the output window. However, when so-called *code-antiquotations* such as `@{code NIL}` are used, the process is more involved because SML antiquotations implicitly refer to Isabelle values. Concretely, there is an additional processing step, resolving the needed Isabelle dependencies before the SML code is actually compiled. declare`[[ML_trace]]` shows this resolving step (varying depending on the antiquoted values). In Fig. 1, the two antiquotations `NIL` and `height` imply the following generated SML code:

```
structure Isabelle =
 struct
   datatype nat  = Zero_nat | Suc  of nat        ;
   datatype list = NIL      | CONS of nat * list ;
   fun height  NIL          = Zero_nat
     | height (CONS (x, t)) = Suc (height t)     ;
 end (*struct Generated_Code*)
```

This SML code reflects equivalently the one of Isabelle side (the datatype and fun declarations). During the compilation, antiquotations are then replaced by:

```
val NIL    = Isabelle.NIL
val height = Isabelle.height
```

which makes "`height NIL`" efficiently executable in the context of the compiled code — no symbolic representation is any longer involved.

Defining Isar Syntax. We are adding a new command Term in Fig. 1 with "`keywords` *Term*"; the *Isar*-component of Isabelle handling the "outer syntax" is in fact reconfigurable. Generally, any command from the Isabelle core APIs is accessible within the ML scope. So it is as well possible to implement Term for simulating datatype, fun or any existing Isabelle command.

In a nutshell, a combination of the techniques shown in this section will be used to construct our compiler in HOL, compile it to SML, and bind the compiled code to a USE tool-like syntax inside Isabelle/Isar.

2.2 UML/OCL: A Running Example

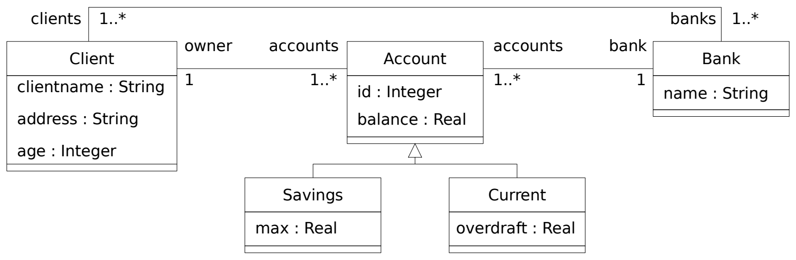


Fig. 2. A simple class model capturing a bank account

Let us consider here a small example of a UML class model together with its class invariants in OCL. The model of Figure 2 describes a set of clients owning bank accounts in different banks. Each account is either a current account or a savings account, and belongs to exactly one bank and one client. A client cannot have more than one current account in a given bank, but as many savings accounts as he likes. If a client is less than 25, his authorised overdraft is 250€ on every of his current accounts, otherwise no overdraft is allowed (it is set to 0). Moreover, the balance of a savings account must be between 0 and `max`. Finally, a consistency constraint has to be imposed: a client owning an account that belongs to a given bank must be a client of this bank.

3 Method: Tool-Construction by Reflection

Although it is perfectly feasible to program the compiler for class models to a corresponding datatype theory in SML , we choose to take even more advantage of the Isabelle framework instead. By using Isabelle/HOL as "implementation language" itself, we profit from the general proof-editing facilities as well as the possibility to *prove* properties over the compiler. For the moment, this covers only termination properties of the compiling functions, but the technique can in principle be used to prove complex meta-theoretic properties such as "if the class model is well-formed, the generated code will be type-correct wrt. HOL types".

The overall structure of the class-model compiler of Featherweight OCL is illustrated in Fig. 3. In the following, we will describe its components.

3.1 UML/OCL Meta-Model

Overall, the compiler has about 6000 lines of code - so we restrict ourselves to critical code-samples in our presentation. As short example, here is how we

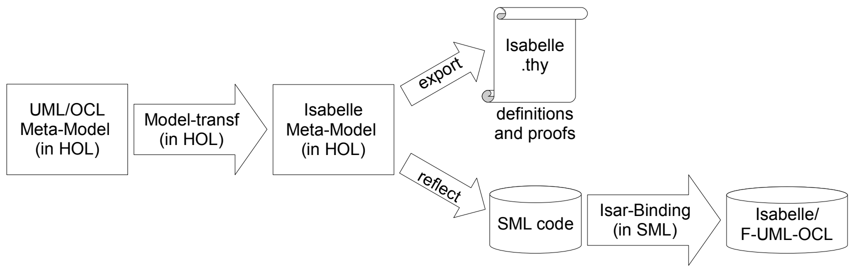


Fig. 3. Overview of the UML/OCL data-model compiler

define the Isabelle/HOL datatype behind the abstract syntax tree of the USE language, that we call the UML/OCL meta-model:

```
datatype uml_class = UmlClass
      string                                          (* name of the class    *)
    ( string (* name *) ∗ uml_ty            ) list  (* attribute            *)
    ( string (* name *) ∗ uml_operation     ) list  (* contract             *)
    ( string (* name *) ∗ string            ) list  (* invariant            *)
      string                                          (* link to superclasses *)
```

In this meta-model, a sample portion of our example Fig. 2 reads as follows:

```
[ UmlClass "Client" [ ( "clientname" , UmlType String),
                      ( "address"    , UmlType String) ] [] [] "OclAny",
  UmlClass "Bank"   [ ( "name"       , UmlType String) ] [] [] "OclAny",
  [...]                                                              ]
```

The compilation proceeds by elementary well-formedness checks and through a semantic elaboration. Finally, definitions for casts from `Client` to `OclAny` were produced, for example, defined in terms of the common data-universe for the given class model, together with proofs for cast-up-cast-down properties as mentioned before. For a complete list of definitions and lemmas, the reader is referred to [4].

3.2 Isabelle Meta-Model

While our meta model for Isabelle/HOL should be generic enough for representing all the concepts occurring in [4], we used a convenient abstraction wrt. to the *real* meta model defined in SML. For the moment, our simplistic abstraction is sufficiently expressive for modeling UML/OCL class models, although a long term goal would be the creation of symbolic tests generation procedures. Here is the Isabelle/HOL meta model describing Isabelle datatypes:

```
datatype hol_datatype =
  Datatype  string                                            (* name         *)
          ( string              (* name      *)
          ∗ hol_simplety list  (* arguments *) ) list        (* constructors *)
```

The previous meta model belongs to the following more general one, while a few is presented, there is actually an abundance of constructors:

```
datatype hol_theory = Theory_datatype  hol_datatype
                    | Theory_definition hol_definition
                    | Theory_lemma     hol_lemma
                    | […]
```

3.3 Model-Transformation and Bindings to Isabelle/jEdit

The transformation from UML-Meta to Isabelle-Meta is purely defined in HOL (no tricks, no axioms, no SML), following linearly the structure of [4]. By applying this generic transformation to the bank example, we obtain automatically the definition of the object universe of [3] instantiated for the bank example: Fig. 4 (top) shows a simple fragment of datatypes (there are also proofs).

As described in Sec. 2.1, we now define a textual format for the UML/OCL meta-model and bind the generated ("reflected") compiler to it. Conceptually, it is similar to the Term command, only that UML data-models are simulated with the raw implementation of the datatype and lemma commands, instead of printing a term. We were inspired by the syntax of the USE tool[7] as entry point in Isabelle/jEdit. Finally, the interactive editing of the bank example leads to 2385 definitions and derived theorems, the source code is in Fig. 4 (bottom).

4 An Empirical Evaluation

This section gives some experimental results on run-time executions of the generated compiler. We study the following scenario: we build a sample of class models, where in each class model, every class inherits *explicitly* from one class, except `OclAny` standing as the only root: thus we have a tree where each class inherits *implicitly* from all classes present in the path of ancestors going to `OclAny`. For example, `Current` is an explicit subclass of `Account`, while `Current` also implicitly inherits from `OclAny`. Attributes and associations between classes are not considered here; as a shorthand, the "subclass" word alone will designate an explicit subclass.

We present Fig. 5 a table reporting the number of theorems associated to each tested class model. Numbers of generated theorems are indicated by powers of 1000 (so Kilo and Mega), and those in italic are an estimation based on the size of the generated file. The class models we are measuring can be identified uniquely by pairs (X, Y) where X is the exact number of subclasses of every class having at least one subclass; and Y is the depth of the inheritance tree (without `OclAny`, so the minimum is one for a tree containing at least `OclAny` with another element). Class-models appear sorted in the table according to the following priority: 1) by row using the total number of classes in the class model, c represents the cardinal without `OclAny`; then 2) by column using the depth of the inheritance tree. For instance, class models in the row $[(1, 30), (2, 4), (5, 2), (30, 1)]$ are sorted in decreasing order by depth, all having 31 classes ($c = 30$, `OclAny` counts for 1).

Since the generated UML/OCL theorems serve for solving (the most automatically) UML/OCL formulas manually entered by the user, our goal is to continue

$$
\begin{aligned}
&\text{datatype } \mathrm{type}_{\mathrm{Savings}} &&= \mathrm{mk}_{\mathrm{Savings}}\ \mathrm{oid}\ \mathrm{int}_{\perp}\ \mathrm{real}_{\perp}\\
&\text{datatype } \mathrm{typeoid}_{\mathrm{Savings}} &&= \mathrm{mkoid}_{\mathrm{Savings}}\ \mathrm{type}_{\mathrm{Savings}}\ \mathrm{real}_{\perp}\\
&\text{datatype } \mathrm{type}_{\mathrm{Account}} &&= \mathrm{mk}_{\mathrm{Account\_Savings}}\ \mathrm{typeoid}_{\mathrm{Savings}}\\
& &&\mid \mathrm{mk}_{\mathrm{Account\_Checks}}\ \mathrm{typeoid}_{\mathrm{Checks}}\\
& &&\mid \mathrm{mk}_{\mathrm{Account}}\ \mathrm{oid}\\
&\text{datatype } \mathrm{typeoid}_{\mathrm{Account}} &&= \mathrm{mkoid}_{\mathrm{Account}}\ \mathrm{type}_{\mathrm{Account}}\ \mathrm{int}_{\perp}\,\mathrm{real}_{\perp}\\
&\text{datatype } \mathrm{type}_{\mathrm{OclAny}} &&= \mathrm{mk}_{\mathrm{OclAny\_Client}}\ \mathrm{typeoid}_{\mathrm{Client}}\\
& &&\mid \mathrm{mk}_{\mathrm{OclAny\_Bank}}\ \mathrm{typeoid}_{\mathrm{Bank}}\\
& &&\mid \mathrm{mk}_{\mathrm{OclAny\_Account}}\ \mathrm{typeoid}_{\mathrm{Account}}\\
& &&\mid \mathrm{mk}_{\mathrm{OclAny\_Savings}}\ \mathrm{typeoid}_{\mathrm{Savings}}\\
& &&\mid \mathrm{mk}_{\mathrm{OclAny\_Checks}}\ \mathrm{typeoid}_{\mathrm{Checks}}\\
& &&\mid \mathrm{mk}_{\mathrm{OclAny}}\ \mathrm{oid}\\
&\text{datatype } \mathrm{typeoid}_{\mathrm{OclAny}} &&= \mathrm{mkoid}_{\mathrm{OclAny}}\ \mathrm{type}_{\mathrm{OclAny}}
\end{aligned}
$$

```
Class Bank
      Attributes
        name        : String
End

Class Client
      Attributes
        clientname : String
        address    : String
        age        : Integer
End

Class Account
      Attributes
        id         : Integer
        balance    : Real
End

Class Savings < Account
      Attributes
        max        : Real
End
```

```
Class Current < Account
      Attributes
        overdraft  : Real
End

Association clients
  Between Bank    [1 .. *]
            Role banks
          Client  [1 .. *]
            Role clients    End

Association accounts
  Between Account [1 .. *]
            Role accounts
          Client  [1]
            Role owner      End

Association bankaccounts
  Between Account [1 .. *]
            Role accounts
          Bank    [1]
            Role bank       End
```

```
Context c: Savings
  Inv A : ‘0 < (c .max)‘
  Inv B : ‘c .balance <= (c .max) and 0 <= (c .balance)‘

Context c: Current
  Inv A : ‘25 < (c .owner .age) implies (c .overdraft = 0)‘
  Inv B : ‘c .owner .age <= 25 implies (c .overdraft = -250)‘

Context c: Client
  Inv A : ‘c .accounts ->collect(banks) = c .banks‘
```

Fig. 4. Modeling the bank account in UML/OCL with Isabelle/jEdit

c	depth c	depth 5	depth 4	depth 3	depth 2	depth 1
12	$(1, c)$ 14K				$(3, 2)$ 12K	$(c, 1)$ 11K
14	$(1, c)$ 20K			$(2, 3)$ 17K		$(c, 1)$ 16K
20	$(1, c)$ *52K*				$(4, 2)$ *39K*	$(c, 1)$ *39K*
30	$(1, c)$ *155K*		$(2, 4)$ *121K*		$(5, 2)$ *115K*	$(c, 1)$ *115K*
39	$(1, c)$ *330K*			$(3, 3)$ *240K*		$(c, 1)$ *240K*
42	$(1, c)$ *409K*				$(6, 2)$ *288K*	$(c, 1)$ *294K*
56	$(1, c)$ *964K*				$(7, 2)$ *649K*	$(c, 1)$ *661K*
62	$(1, c)$ *1.3M*	$(2, 5)$ *907K*				$(c, 1)$ *882K*
72	$(1, c)$ *2M*				$(8, 2)$ *1.3M*	$(c, 1)$ *1.3M*
84	$(1, c)$ *3.3M*			$(4, 3)$ *2.1M*		$(c, 1)$ *2.1M*
90	$(1, c)$ *4.2M*				$(9, 2)$ *2.5M*	$(c, 1)$ *2.5M*

Fig. 5. Number of theorems generated

to generate new UML/OCL theorems, while keeping an objective of shortening the size of proofs whenever applicable. In a class model with only `OclAny` as class, we obtain a theory comprising 182 theorems proven automatically; by adding another class, we reach 384 theorems.

Besides the constraint on the generation of a high number of theorems, time or space involved for performing the generation is obviously a criteria to consider as enhancement: we need at least 9G of RAM memory for generating in 1 min one of the three examples of $c = 56$ classes. For $c = 90$, 28G becomes mandatory to be executed in 7 min; however, there are substantial potentials for optimization.

5 Conclusion

We have shown a method to construct semantic-based tools for textual domain-specific languages (DSLs) based on UML and OCL. Based on Isabelle/HOL theories that capture the semantic essence of a DSL (in our case: class-models plus OCL invariants and contracts), we describe model-transformation from a formal UML meta-model to an Isabelle meta-model that generates the necessary properties automatically. Compared to conventional implementations of *code-generators* for OCL, the resulting tool is clearly not competitive in terms of compilation size of models, essentially because each theorem is complemented with a proof of a certain size. On the other hand our tool is unique that it actually generates a large number of theorems resulting from class-models which are necessary for symbolic execution and UML/OCL interactive proving. Moreover, our tool — for which we still see a large potential of optimization — can serve as reference environment for the UML/OCL language.

Related Work. The idea to use SML for supporting data-type theories is in itself very old and deeply linked from the very beginning with theorem proving environments such as Edinburgh-LCF, HOL4, HOL-light, Isabelle and Coq. The application to *object-oriented* data-type theories is also not new — earlier works in this line are [11], for example. In contrast to [5], we applied these techniques to UML under *closed-world assumption* for a standard-conform 4-valued logics for OCL, which is seen as the semantic framework for DSL's. This is particularly important and challenging since heterogenous system specifications need to be combined in a seamless way, and since semantically correct tools have to be developed for these language combinations.

Future Work. It is our ultimate goal to develop the technology up to the point that it can be used for automated test-generation following the lines of [2]. We expect that the approach can be applied to textually presented sequence models, state-machines or MARTE-profiles.

Acknowledgments. This research work has been carried out under the leadership of the Technological Research Institute SystemX, and therefore granted with funds of the Program "Investissements d'Avenir" (`www.irt-systemx.fr`).

References

[1] U. Aßmann, A. Bartho, C. Bürger, S. Cech, B. Demuth, F. Heidenreich, J. Johannes, S. Karol, J. Polowinski, J. Reimann, J. Schroeter, M. Seifert, M. Thiele, C. Wende, and C. Wilke. Dropsbox: the dresden open software toolbox. *Software & Systems Modeling*, 13(1):133–169, 2014.

[2] A. D. Brucker, M. P. Krieger, D. Longuet, and B. Wolff. A specification-based test case generation method for UML/OCL. In J. Dingel and A. Solberg, editors, *MoDELS Workshops*, number 6627, pages 334–348. 2010. Selected best papers from all satellite events of the MoDELS 2010 conference. Workshop on OCL and Textual Modelling.

[3] A. D. Brucker, D. Longuet, F. Tuong, and B. Wolff. On the semantics of object-oriented data structures and path expressions. In J. Cabot, M. Gogolla, I. Rath, and E. Willink, editors, *Workshop on OCL, Model Constraint and Query Languages (OCL 2013)*, 2013. An extended version of this paper is available as **lri!** Technical Report 1565.

[4] A. D. Brucker, F. Tuong, and B. Wolff. Featherweight OCL: A proposal for a machine-checked formal semantics for OCL 2.5. *Archive of Formal Proofs*, Jan. 2014. `http://afp.sf.net/entries/Featherweight_OCL.shtml`, Formal proof development.

[5] A. D. Brucker and B. Wolff. An Extensible Encoding of Object-oriented Data Models in HOL with an Application to IMP++. *Journal of Automated Reasoning (JAR)*, Selected Papers of the AVOCS-VERIFY Workshop 2006(3–4):219–249, 2008. Serge Autexier, Heiko Mantel, Stephan Merz, and Tobias Nipkow (eds).

[6] A. D. Brucker and B. Wolff. Featherweight OCL: A study for the consistent semantics of OCL 2.3 in HOL. In *Workshop on OCL and Textual Modelling (OCL 2012)*, pages 19–24, 2012. The semantics for the Boolean operators proposed in this paper was adopted by the OCL 2.4 standard.

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

[8] T. Parr. *The Definitive ANTLR Reference Guide: Building Domain-Specific Languages.* Pragmatic Programmers, 2007.

[9] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks. *EMF: Eclipse Modeling Framework, 2nd Edition.* Eclipse Series. Addison-Wesley Professional, Dec 16, 2008.

[10] The Object Management Group (OMG). The MOF and OMG's model driven architecture (MDA). Available at `http://www.omg.org/mof` (2014).

[11] M. Wenzel. Type classes and overloading in higher-order logic. In *TPHOLs*, pages 307–322, 1997.

[12] E. Willinks. The Eclipse OCL project. Available at `http://www.willink.me.uk` (2014).