Featherweight OCL

A study for the consistent semantics of OCL 2.3 in HOL

Achim D. Brucker
SAP AG, SAP Research
Vincenz-Priessnitz-Str. 1
76131 Karlsruhe, Germany
achim.brucker@sap.com

Burkhart Wolff*
Université Paris-Sud
Parc Club Orsay Université
91893 Orsay Cedex, France
wolff@lri.fr

ABSTRACT

At its origins, OCL was conceived as a strict semantics for undefinedness, with the exception of the logical connectives of type `Boolean` that constitute a three-valued propositional logic. Recent versions of the OCL standard added a second exception element, which, similar to the null references in programming languages, is given a non-strict semantics.

In this paper, we report on our results in formalizing the core of OCL in higher-order logic (HOL). This formalization revealed several inconsistencies and contradictions in the current version of the OCL standard. These inconsistencies and contradictions are reflected in the challenge to define and implement OCL tools in a uniform manner.

Categories and Subject Descriptors

D3.1.1 [**Software**]: Programming Languages—*Formal Definitions and Theory*

1. INTRODUCTION

At its origins [13, 16], OCL was conceived as a strict semantics for undefinedness, with the exception of the logical connectives of type `Boolean` that constitute a three-valued propositional logic. Recent versions of the OCL standard [14, 15] added a second exception element, which is given a non-strict semantics. Unfortunately, this extension results in several inconsistencies and contradictions. These problems are reflected in difficulties to define interpreters, code-generators, specification animators or theorem provers for OCL in a uniform manner and resulting incompatibilities of various tools. For the OCL community, this results in the challenge to define a new formal semantics definition OCL that could replace the "Annex A" of the OCL standard [15].

In the paper "Extending OCL with Null-References" [8] we explored—based on mathematical arguments and paper and pencil proofs—a consistent formal semantics that comprises two exception elements: `invalid` ("bottom" in semantics terminology) and `null` (for "non-existing element").

*This work was partly supported by the Digiteo Foundation.

This short paper is based on a formalization of [8], called "Featherweight OCL," in Isabelle/HOL [12]. This formalization is in its present form merely a semantical study and a proof of technology than a real tool. It focuses on the formalization of the key semantical constructions, i. e., the type `Boolean` and the logic, the type `Integer` and a standard strict operator library, and the collection type `Set(A)` with quantifiers, iterators and key operators.

The rest of this paper summarizes our experiences and findings in formalizing a core of OCL 2.3 in Isabelle/HOL. Thus, this paper serves as an extended abstract of the detailed documents that are available at `http://www.brucker.ch/projects/hol-ocl/Featherweight-OCL/`.

2. FEATHERWEIGHT OCL

Featherweight OCL is a formalization of the core of OCL aiming at formally investigation the relationship between the different notions of "undefinedness," i. e., `invalid` and `null`. As such, it does not attempt to define the complete OCL library. Instead, it concentrates on the core concepts of OCL as well as the types `Boolean`, `Integer`, and typed sets (`Set(T)`). Following the tradition of HOL-OCL [3, 5], Featherweight OCL is based on the following principles:

1. It is an embedding into a powerful semantic meta-language and environment, namely Isabelle/HOL [12].
2. It is a *shallow embedding* in HOL; types in OCL were injectively mapped to types in Featherweight OCL. Ill-typed OCL specifications cannot be represented in Featherweight OCL and a type in Featherweight OCL contains exactly the values that are possible in OCL. Thus, sets may contain `null` (`Set{null}` is a defined set) but not `invalid` (`Set{invalid}` is just `invalid`).
3. Any Featherweight OCL type contains at least `invalid` and `null` (the type `Void` contains only these instances). The logic is consequently four-valued, and there is a `null`-element in the type `Set(A)`.
4. It is a strongly typed language in the Hindley-Milner tradition. We assume that a pre-process eliminates all implicit conversions due to subtyping by introducing explicit casts (e. g., `oclAsType()`). The details of such a pre-processing are described in [2]. Casts are semantic functions, typically injections, that may convert data between the different Featherweight OCL types.
5. All objects are represented in an object universe in the HOL-OCL tradition [4] the universe construction also gives semantics to type casts, dynamic type tests, as well as functions such as `oclAllInstances()`, or `isNewInState()`.

6. Featherweight OCL types may be arbitrarily nested: `Set{Set{1,2}} = Set{Set{2,1}}` is legal and true.
7. For demonstration purposes, the set-type in Featherweight OCL may be infinite, allowing infinite quantification and a constant that contains the set of all Integers. Arithmetic laws like commutativity may therefore expressed in OCL itself. The iterator is only defined on finite sets.
8. It supports equational reasoning and congruence reasoning, but this requires a differentiation of the different equalities like strict equality, strong equality, meta-equality (HOL). Strict equality and strong equality require a subcalculus, "cp" (a detailed discussion of the different equalities as well the subcalculus "cp"—for three-valued OCL 2.0—is given in [7]), which is nasty but can be hidden from the user inside tools.

3. LESSONS LEARNED

While our paper and pencil arguments, given in [8], turned out to be essentially correct, there had also been a lesson to be learned: If the logic is not defined as a Kleene-Logic, having a structure similar to a complete partial order (CPO), reasoning becomes complicated: several important algebraic laws break down which makes reasoning in OCL inherent messy and a semantically clean compilation of OCL formulae to a two-valued presentation, that is amenable to animators like KodKod [17] or SMT-solvers like Z3 [10] completely impractical. Concretely, if the expression `not(null)` is defined `invalid` (as is the case in the present standard [15]), than standard involution does not hold, i. e., `not(not(A))` = `A` does not hold universally. Similarly, if `null and null` is `invalid`, then not even idempotence `X and X` = `X` holds. We strongly argue in favor of a lattice-like organization, where `null` represents "more information" than `invalid` and the logical operators are monotone with respect to this semantical "information ordering."

Featherweight OCL makes these two deviations from the standard, builds all logical operators on Kleene-`not` and Kleene-`and`, and shows that the entire construction of our paper "Extending OCL with Null-References" [8] is then correct, and the DNF-normaliation as well as δ-closure laws (necessary for a transition into a two-valued presentation of OCL specifications ready for interpretation in SMT solvers (see [9] for details) are valid in Featherweight OCL.

4. CONCLUSION AND FUTURE WORK

Featherweight OCL concentrates on formalizing the semantics of a core subset of OCL in general and in particular on formalizing the consequences of a four-valued logic (i. e., OCL versions that support, besides the truth values `true` and `false` also the two exception values `invalid` and `null`).

In the following, we outline the necessary steps for turning Featherweight OCL into a fully fledged tool for OCL, e. g., similar to HOL-OCL as well as for supporting test case generation similar to HOL-TestGen [6]. There are essentially five extensions necessary:

- extension of the library to support all OCL data types, e. g., `Sequence(T)`, `OrderedSet(T)`. This formalization of the OCL standard library can be used for checking the consistency of the formal semantics (known as "Annex A") with the informal and semi-formal requirements in the normative part of the OCL standard.
- development of a compiler that compiles a textual or CASE tool representation (e. g., using XMI or the textual syntax of the USE tool [16]) of class models. Such compiler could also generate the necessary casts when converting standard OCL to Featherweight OCL as well as providing "normalizations" such as converting multiplicities of class attributes to into OCL class invariants.
- a setup for translating Featherweight OCL into a two-valued representation as described in [9]. As, in real-world scenarios, large parts of UML/OCL specifications are defined (e. g., from the default multiplicity `1` of an attributes `x`, we can directly infer that for all valid states `x` is neither `invalid` nor `null`), such a translation enables an efficient test case generation approach.
- a setup in Featherweight OCL of the Nitpick animator [1]. It remains to be shown that the standard, Kodkod [17] based animator in Isabelle can give a similar quality of animation as the OCLexec Tool [11]
- a code-generator setup for Featherweight OCL for Isabelle's code generator. For example, the Isabelle code generator supports the generation of F#, which would allow to use OCL specifications for testing arbitrary .net-based applications.

The first two extensions are sufficient to provide a formal proof environment for OCL 2.3 similar to HOL-OCL while the remaining extensions are geared towards increasing the degree of proof automation and usability as well as providing a tool-supported test methodology for UML/OCL.

Our work shows that developing a machine-checked formal semantics of recent OCL standards still reveals significant inconsistencies—even though this type of research is not new. In fact, we started our work already with the 1.x series of OCL. The reasons for this ongoing consistency problems of OCL standard are manifold. For example, the consequences of adding an additional exception value to OCL 2.2 are widespread across the whole language and many of them are also quite subtle. Here, a machine-checked formal semantics is of great value, as one is forced to formalize all details and subtleties. Moreover, the standardization process of the OMG, in which standards (e. g., the UML infrastructure and the OCL standard) that need to be aligned closely are developed quite independently, are prone to adhoc changes that attempt to align these standards. And, even worse, updating a standard document by voting on the acceptance (or rejection) of isolated text changes does not help either. Here, a tool for the editor of the standard that helps to check the consistency of the whole standard after each and every modifications can be of great value as well.

References

[1] J. C. Blanchette and T. Nipkow. Nitpick: A counterexample generator for higher-order logic based on a relational model finder. In M. Kaufmann and L. C. Paulson, editors, *ITP*, LNCS 6172, pages 131–146. Springer, 2010. doi: 10.1007/978-3-642-14052-5_11.

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

[3] A. D. Brucker and B. Wolff. The HOL-OCL book. Technical Report 525, ETH Zurich, 2006.

[4] A. D. Brucker and B. Wolff. An extensible encoding of object-oriented data models in HOL. *Journal of Au-*

tomated Reasoning, 41:219–249, 2008. doi: 10.1007/s10817-008-9108-3.

[5] A. D. Brucker and B. Wolff. HOL-OCL – A Formal Proof Environment for UML/OCL. In J. Fiadeiro and P. Inverardi, editors, *FASE*, LNCS 4961, pages 97–100. Springer, 2008. doi: 10.1007/978-3-540-78743-3_8.

[6] A. D. Brucker and B. Wolff. HOL-TestGen: An interactive test-case generation framework. In M. Chechik and M. Wirsing, editors, *FASE*, LNCS 5503, pages 417–420. Springer, 2009. doi: 10.1007/978-3-642-00593-0_28.

[7] A. D. Brucker and B. Wolff. Semantics, calculi, and analysis for object-oriented specifications. *Acta Informatica*, 46(4): 255–284, 2009. doi: 10.1007/s00236-009-0093-8.

[8] A. D. Brucker, M. P. Krieger, and B. Wolff. Extending OCL with null-references. In S. Gosh, editor, *Models in Software Engineering*, LNCS 6002, pages 261–275. Springer, 2009. doi: 10.1007/978-3-642-12261-3_25.

[9] 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*, LNCS 6627, pages 334–348. Springer, 2010. doi: 10.1007/978-3-642-21210-9_33.

[10] L. M. de Moura and N. Bjørner. Z3: An efficient SMT solver. In C. R. Ramakrishnan and J. Rehof, editors, *TACAS*, LNCS 4963, pages 337–340. Springer, 2008. doi: 10.1007/978-3-540-78800-3_24.

[11] M. P. Krieger, A. Knapp, and B. Wolff. Generative programming and component engineering. In E. Visser and J. Järvi, editors, *GPCE*, pages 53–62. ACM, 2010.

[12] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL—A Proof Assistant for Higher-Order Logic*, LNCS 2283. Springer, 2002. doi: 10.1007/3-540-45949-9.

[13] Object Management Group. Object constraint language specification (version 1.1), Sept. 1997. Available as OMG document ad/97-08-08.

[14] Object Management Group. UML 2.0 OCL specification, Apr. 2006. Available as OMG document formal/06-05-01.

[15] Object Management Group. UML 2.3.1 OCL specification, Feb. 2012. Available as OMG document formal/2012-01-01.

[16] M. Richters. *A Precise Approach to Validating UML Models and OCL Constraints.* PhD thesis, Universität Bremen, Logos Verlag, Berlin, BISS Monographs, No. 14, 2002.

[17] E. Torlak and D. Jackson. Kodkod: A relational model finder. In O. Grumberg and M. Huth, editors, *TACAS*, LNCS 4424, pages 632–647. Springer, 2007. doi: 10.1007/978-3-540-71209-1_49.