Formalizing UML Models and OCL Constraints in PVS [1]

Marcel Kyas and Harald Fecher [2]

Institute for Computer Science and Applied Mathematics, Christian-Albrechts-Universität zu Kiel, Germany

Frank S. de Boer and Joost Jacob [3]

CWI Amsterdam, The Netherlands

Jozef Hooman and Mark van der Zwaag [4]

Computer Science Department, University of Nijmegen, The Netherlands

Tamarah Arons and Hillel Kugler [5]

Weizmann Institute of Science, Rehovot, Israel

Abstract

The Object Constraint Language (OCL) is the established language for the specification of properties of objects and object structures in UML models. One reason that it is not yet widely adopted in industry is the lack of proper and integrated tool support for OCL. Therefore, we present a prototype tool, which analyzes the syntax and semantics of OCL constraints together with a UML model and translates them into the language of the theorem prover PVS. This defines a formal semantics for both UML and OCL, and enables the formal verification of systems modeled in UML. We handle the problematic fact that OCL is based on a three-valued logic, whereas PVS is only based on a two valued one.

Key words: OCL, PVS, Formal Verification, Formal Semantics, UML

[1] Part of this work has been financially supported by IST project Omega (IST-2001-33522), see `http://www-omega.imag.fr`, and NWO/DFG project Mobi-J (RO 1122/9-1, RO 1122/9-2).

[2] `mailto:{mky,hf}@informatik.uni-kiel.de`

[3] `mailto:{frb,jacob}@cwi.nl`

[4] `mailto:{hooman,mbz}@cs.kun.nl`

[5] `mailto:{tamarah,kugler}@wisdom.weizmann.ac.il`

1 Introduction

Today, UML [7,8] and its textual specification language OCL [11,1] are widely used as specification and modeling languages for object-oriented systems. A wide range of tools are available supporting the development of systems using UML's notations, from simple syntactic analyzers to simulators, compilers enabling run-time checking of specifications, model checkers, and theorem provers. However, there exists no program that integrates verification and validation of UML class diagrams, state machines, and OCL specifications.

We focus on deductive verification in higher order logic. This allows the verification of possibly infinite state systems. We present a translation of the notations of UML (class diagrams, state machines, and OCL constraints) into the input language of PVS [9]. Then the specification, originally given in OCL, can be verified using PVS.

The compiler we describe does implements a translation of a well-defined subset of UML diagrams which is sufficient for many applications. This subset consists of class diagrams which only have associations with a multiplicity of 0 or 1 and no generic classes, flat state-machines (state-machines can always be represented as a flat state machine with the same behavior, as described in [13]), and OCL constraints.

OCL, a three-valued logic, has then to be encoded in PVS, which is based on a two-valued logic and which only allows total functions. The reason for OCL's three-valuedness is that it uses partial functions and that relations are interpeted as strict functions into the three truth-values. Furthermore, the additional truth value only occurs indirectly by applying a function to a value outside its domain. The way partial functions of OCL are translated to PVS decides how the three-valued logic is handled. Our transformation restricts a partial function to its domain yielding a total function.

The work reported in [12] defines a formalization of UML state machines in PVS[6], which does not include a translation of OCL to PVS. In [2] a formal semantics of OCL has been proposed in the theorem-prover Isabelle/HOL [6]. Contrary to our approach, partial functions have been extended to total functions by introducing an undefined value. While this approach allows the verification of meta theorems, the verification of an actual UML model w.r.t. its OCL specification still requires the additional proof that all values are defined.

2 Running Example

We use the *Sieve of Eratosthenes* as a running example. It is modeled using the two classes *Generator* and *Sieve* (see Figure 1). Exactly one instance, the root object, of the class Generator is present in the model. The generator creates an instance of the Sieve class. Then it sends the new instance natural

[6] It has been implemented in the PrUDE program, see `http://www.isot.ece.uvic.ca/download.html`.

numbers in increasing order, see Figure 2. The association from Generator to Sieve is called *itsSieve*.

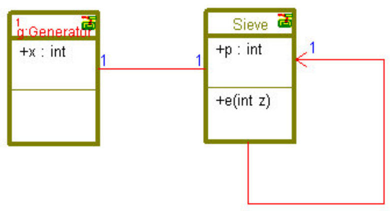

Fig. 1. Class Diagram of the Sieve Example

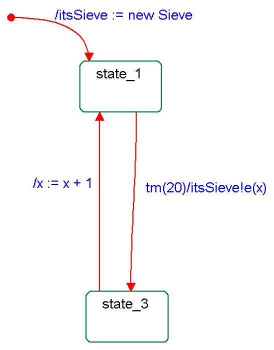

Fig. 2. State Machine of the Generator

Upon creation, each instance of Sieve receives a prime number and stores it in its attribute p. Then it creates a successor object, called *itsSieve* [7], and starts receiving a sequence of integers i. If p divides i, then this instance does nothing. Otherwise it sends i to *itsSieve*. This behavior is shown in Figure 3. [8]

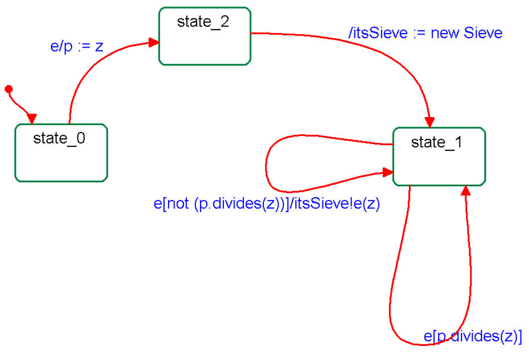

Fig. 3. State Machine of a Sieve

The safety property we would like to prove is that p is a prime number for each instance of Sieve; this can be formalized in OCL by:

```
context Sieve inv: Integer{2..(p-1)}->forAll(i | p.mod(i) <> 0)
```

This constraint states that the value of the attribute p is not divisible by any number i between 2 and $p - 1$. To prove this, we need to establish, that the sequence of integers received by each instance of Sieve is monotonically increasing.

We have chosen this example, because it is short, but still challenging to verify. It involves object creation and asynchronous communication, and therefore does not have a finite state space. Furthermore, the behavior of the

[7] The association from the Sieve class to itself in Figure 1is called *itsSieve*.
[8] The trigger *tm(20)* in Figure 2 postpones its reaction by 20 time units.

model depends on the data sent between objects and note also that the property we want to prove on the model is a number-theoretic property, namely, that the numbers generated are primes. This makes it impossible to show the considered property using automatic techniques like model checking.

3 Representation in PVS

The UML diagrams we considere are restricted forms of class diagrams and state machines, which both may contain OCL expressions.

3.1 Class Diagrams

To formalize class diagrams in PVS we define a type `Class` which enumerates the names of all classes appearing in the model and a predicate on classes which states whether the class is `active`. The constant `rootClass` denotes the active class which provides the first object in the model. Furthermore, the attribute, operation, signal, and reference names of each class are enumerated as a type. The association of a name to its defining class is done by prefixing the name with the corresponding class name. The translation of the class diagram shown in Figure 1 is presented in Figure 4.

```
Class: TYPE+ = { Generator, Sieve }
active: pred[Class] = LAMBDA (c: Class):
  c = Generator OR c = Sieve;
rootClass: (active) = Generator
Attribute: TYPE+ = { Generator___x,
  Sieve___z, Sieve___p, unusedAttribute }
Reference: TYPE+ = { Sieve___itsSieve,
  Generator___itsSieve, Sieve___itsGenerator,
  unusedReference }
```

Fig. 4. Translation of the Sieve Class Diagram

```
Location: TYPE+ = { Generator____61,
  Generator____64, Generator____66,
  Sieve____25, Sieve____28,
  Sieve____32, Sieve____33 }
t28: Transition = (#
  source := Sieve____28,
  trigger := signalEvent(Sieve___e, Sieve___z),
  guard := (LAMBDA (val:
    Valuation): (NOT
      divides((val`aval(Sieve___p)),
    (val`aval(Sieve___z))))),
  actions := (cons((emitSignal(Sieve___itsSieve,
    Sieve___e, (LAMBDA (val: Valuation):
      (val`aval(Sieve___z))))), null)),
  target := Sieve____28,
  class := Sieve
#);
transitions: setof[Transition] = { t: Transition |
  t = t9 OR t = t11 OR t = t13 OR
  t = t25 OR t = t28 OR t = t32 OR
  t = t34 OR t = t37 }
```

Fig. 5. Translation of the Generator State Machine

Objects and object structures are always obtained from these definitions using interpretation functions. We assume a PVS type `Object`, whose elements represent all objects. The function `class: [Object -> Class]` assigns each object its class and the function `state: [Object -> [Attribute -> Value]]` assigns each object its state, which is a valuation of all attributes. Type checking asserts that all objects only uses attributes defined for their class, so we assign to the attributes not defined in an objects' class an arbitrary value.

3.2 State Machines

State machines are represented as graphs and OCL expressions occuring in the state machine are translated to expressions representing their semantics. The semantics of the translated state machines is given in terms of sets of computations using a function like the one described in [5]. For example, the transition from *state_1* to *state_1* sending an integer to the next object, as shown in Figure 2, is translated to the PVS fragment shown in Figure 5.

3.3 OCL

OCL is a three-valued logic that assigns to each formula a value *true*, *false*, or *undefined* ($\perp$). The reasons for this are that each partial function f is extended in the semantics of OCL to a *strict* total function $f_\perp \overset{def}{=} \lambda x.\mathsf{if}\ x \in \mathrm{dom}(f)\ \mathsf{then}\ f(x)\ \mathsf{else}\ \perp\ \mathsf{fi}$ and that relations and predicates are interpreted as strict functions to the three truth-values [11].

The OCL standard, however, states that a model is well-formed only if all constraints are *true*. Since we are interested in verifying the correctness of a model w.r.t. its OCL specification, we have to prove that all OCL constraints are *true*, i.e., neither *false* nor *undefined*. To achieve this, each OCL formula is translated directly to PVS, such that we equate the truth values *false* and $\perp$. The advantages of this approach are: (1) We do not need to require that each function is strict in its arguments. PVS provides its own method through the automatic generation of *type consistency constraints* (TCC) to handle this. (2) We do not need to redefine the core of the logic, e.g., the *and* and *implies* functions, in PVS. Instead, use is made of PVS's strategies.

In order to reduce OCL's semantics to a two-valued one, we have to consider three situations carefully: partial primitive functions, the operational flavor of OCL's semantics, and undefined values in the state.

Some primitive functions used in OCL are partial functions which may return an undefined value, e.g., division by zero. In [2] Brucker et al. have extended the partial functions to total functions by explicitly introducing the undefined value and formalizing the underlying three-valued logic, which is closer to the semantics of OCL. The disadvantage of this approach is that reasoning in a three-valued logic loses PVS's automation, mostly because the law of the excluded middle does not hold in OCL, and causes the redefinition of all predefined functions of PVS.

Our approach is to *restrict* each partial function to its domain, which makes it a total function. This requires a formalization of the domains of each primitive function. Many functions used in OCL already have a corresponding equivalent one in PVS, e.g., the arithmetic functions. The missing functions have been defined as total functions by us in a library, with an appropriate semantics.

OCL allows user-defined functions in expressions, which are either introduced using a let-construct or by using an operation that has been declared side-effect free. Provided that an implementation of the user-defined function is given, we compute a signature that is based on type definitions computed from the class hierarchy. All instances in a model are identified by a value of the type `OclAny`[9]. It contains the special literal `nil`[10] which represents the non-existing object. In PVS we define the type `ObjectNotNil` containing all existing objects excluding `nil`. For each class `Class` defined in the model we introduce a type `ObjectClass` and generate a subtype `ObjectClassNotNil` of `ObjectClass`. If a class `Class2` is a direct subclass of `Class` then the super-type of `ObjectClass2` is `ObjectClass` in PVS. This encodes the usual interpretation of the input's class hierarchy in PVS and satisfies the subsumption property that if a class C is a subclass of D, then each instance of C is also an instance of D.

The signature of a user-defined operation in PVS is obtained directly from the original signature, except that the type of the first argument that is always *self*, becomes `ObjectClassNotNil`, in which `Class` refers to the name of the class in which the operation is defined. Then PVS generates TCCs to assert that all arguments are in the domain of a function. A failure to prove them in most cases indicates that the original OCL expression is either false or undefined.

The formal semantics of OCL is concerned with *executing* OCL constraints. As an effect the value of an recursive function is undefined, if its evaluation is diverging. This semantics may be suitable for run-time checking, but it is not implementable, because the termination problem is generally undecidable. In PVS, however, the termination of every recursive function has to be guaranteed by a *ranking function* and a termination proof. Therefore, we translate recursive functions of OCL to recursive functions in PVS directly. The user has to define the ranking function in the PVS output himself, because OCL does not provide any means to define such ranking functions.

The semantics of OCL defines the meaning of universal and existential quantification as an possibly infinite expansion of conjunctions, resp. disjunctions. Together with the `allInstances()` operation, whose intuitive meaning is the set of all existing class' instances, this allows counter-intuitive specifications: The evaluation of the expression `Integer.allInstances()` results in the set of all integers, which is an infinite set. Hence, the expression `Integer.allInstances()->forAll(n | true)` will not terminate in OCL's semantics, and has, therefore, the value *undefined*. Our translation ignores this complication and translates the quantified expressions directly to the PVS ex-

[9] The class `OclAny` is the superclass of all classes in a model, similar to Java's `Object` class.
[10] Note that `nil` is well-defined and represents any empty collection.

pression `FORALL (n: int): TRUE`. The main advantage of this approach is that specifications are much easier to prove. We loose soundness of the translation by this choice, because the translated constraint is provable in PVS but its original OCL constraint is undefined, nevertheless we favor our semantics, as it usually better reflects the user's intention. We seriously doubt, as many others, that the designers of OCL have chosen the right interpretation for the `allInstances()` function and quantification.

Finally, the third way to write specifications which are undefined in the semantics of OCL is by accessing undefined values in the object diagram (or state) of a model, e.g., by accessing an array member outside of the bounds of the array, accessing an attribute not defined (but declared) in this class, or retyping (casting) an object to a subtype of its real type. In a real programming language this usually leads to a run-time exception. We assume that the underlying behavioral semantics guarantees that every attribute is defined, and generate suitable assumptions for the other cases, because we are mainly interested in partial correctness.

4 Initial Experience

The formalism and the methods described in this paper have been implemented in a prototype. We have tested the compiler by, e.g., verifying the object-oriented version of the "Sieve of Eratosthenes" described in Sec. 2. We have proved the safety property that only prime numbers are generated. The proof uses TL-PVS [10].

The complexity of the transition relation generated by our compiler proved to be challenging. It appears that this complexity is inherent to the UML semantics. The most difficult part is reasoning about messages in queues. The concepts of messages preceding one another, crucial for the sieve, are difficult to work with for purely technical reasons. The proof of the sieve depends on the facts that no signals are ever discarded and that signals are taken from the queue in a first-in-first-out order. These two properties have to be specified in PVS as invariants and proved separately.[11] Note, that, if one of the two properties does not hold, then the sieve would not satisfy its specification.

The run-time of our compiler is usually less than a minute, but it is in any case dominated by the time required to prove the model correct in PVS. The coarse level of specifications in OCL are not sufficient to automate the whole verification process. For the proofs, annotations of the states of a state machine expressing invariants might be highly useful, as these have to be formulated as intermediate steps in the current proof. This extension entails some changes of the semantic representation, as the proof method resulting from this is more similar to Floyd's inductive assertion networks (see [4] for

[11] This is not a limitation of PVS but a limitation of interactive theorem proving in general.

references) as implemented in [3].

References

[1] Boldsoft and Rational Software Corporation and IONA and Adaptive Ltd., "Response to the UML 2.0 OCL RfP (ad/2000-09-03)," (2003), revised Submission, Version 1.6 (ad/2003-01-07).

[2] Brucker, A. D. and B. Wolff, *A proposal for a formal OCL semantics in Isabelle/HOL*, in: V. Carreño, C. Muñoz and S. Tashar, editors, *TP-HOL '02*, number 2410 in LNCS (2002), pp. 99–114.

[3] de Boer, F. S. and C. Pierik, *Computer-aided specification and verification of annotated object-oriented programs*, in: A. Rensink and B. Jacobs, editors, *FMOODS '02* (2002), pp. 163–177.

[4] de Roever, W.-P., F. S. de Boer, U. Hannemann, J. Hooman, Y. Lakhnech, M. Poel and J. Zwiers, "Concurrency Verification," Cambridge University Press, 2001.

[5] Hooman, J. and M. van der Zwaag, *A semantics of communicating reactive objects with timing*, in: *SVERTS*, 2003, `http://www-verimag.imag.fr/EVENTS/2003/SVERTS/PAPERS-WEB/13-HoomanZwaag.pdf`.

[6] Nipkow, T., L. C. Paulson and M. Wenzel, "Isabelle/HOL – A Proof Assistant for Higher-Order Logic," Number 2283 in LNCS, Springer-Verlag, 2002.

[7] Object Management Group, "UML 2.0 Infrastructure Specification," (2003).

[8] Object Management Group, "UML 2.0 Superstructure Specification," (2003).

[9] Owre, S., J. Rushby and N. Shankar, *PVS: A prototype verification system*, in: D. Kapur, editor, *CADE '92*, number 607 in LNAI (1992), pp. 748–752.

[10] Pnueli, A. and T. Arons, *TLPVS: A PVS-based LTL verification system*, in: N. Dershowitz, editor, *Proceedings of the International Symposium on Verification – Theory and Practice – Honoring Zohar Manna's 64th Birthday*, number 2772 in LNCS (2003).

[11] Richters, M., "A Precise Approach to Validating UML Models and OCL Constraints," Ph.D. thesis, Universtät Bremen (2002).

[12] Traoré, I., *An outline of PVS semantics for UML statecharts*, Journal of Universal Computer Science **6** (2000), pp. 1088–1108.

[13] Varró, D., *A formal semantics of UML statecharts by model transition systems*, in: A. Corradini, H. Ehrig, H.-J. Kreowski and G. Rozenberg, editors, *ICGT '02*, number 2505 in LNCS (2002), pp. 378–392.