Verifying OCL specifications of UML models : tool support and compositionality

Kyas, M.

Chapter 4

Formalising UML Models and OCL Constraints in PVS

The Object Constraint Language (OCL) is not yet widely adopted in industry, because proper and integrated tool support is lacking. We describe a prototype tool, which analyses 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 modelled 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.

4.1 Introduction

Today, many tools are available which support developing systems using UML's notations, ranging from syntactic analysers (Warmer [153]) to simulators (OCLE [32]), compilers enabling run-time checking of specifications (Hussmann, Demuth, and Finger [68]), model checkers (Latella, Majzik, and Massnik [92] or del Mar Gallardo, Merino, and Pimentel [47]), and integrations with theorem provers (Aredo [6]). However, until now no tool integrates verification and validation of UML class diagrams, state machines, and OCL specifications.

In this chapter we describe a compiler that 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 behaviour, as described by Varro in [150]), and OCL constraints.

We focus on deductive verification in higher-order logic. This allows the verification of possibly infinite-state systems. Therefore, we describe a translation of a subset of the UML into the input language of the interactive theorem prover PVS [121]. This enables the verification of the specification, originally given in OCL, using PVS.

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 interpreted as strict functions into the three truth-values. Furthermore, the additional truth value only occurs by applying a function to a value outside its domain, but it is not represented by a literal.

The way partial functions of OCL are translated to PVS decides how the three-valued logic is handled. Our transformation restricts a partial function, as they occur in OCL specifications, to its domain, yielding a total function. Note that because most functions defined in specifications do not include recursive calls, the restricted domain can be easily computed. Recursive functions, which occur by way of translating recursive definitions in OCL, have to be modified by the user anyway, because in general we cannot prove that these definitions are defined for any input.

Then PVS generates proof obligations, so called *type consistency constraints* (TCC), which establish the type correctness of a PVS expression, in this case, that the function is defined for the declared domain, by way of a proof. If this fails, the user can always correct the generated output or change his constraints.[1]

This chapter is structured as follows. In Section 4.2 we describe the overall approaches of embedding a notation into a theorem prover and motivate our choice of embedding. In Section 4.3 we give a short introduction into the PVS specification language. In Section 4.4 we introduce an example used to illustrate how the translator is working. In Section 4.5 we describe the method used to formalise the input language in PVS and how our two-valued semantics of OCL is obtained. In Section 4.6 we argue that the translation is sound. Finally, in Section 4.7 we report on our initial experience with the described tool, draw conclusions, and report on related work.

4.2 Shallow versus Deep Embedding

If one translates a UML model with its OCL constraints into the logic of a theorem prover, one has to solve the problem how the different languages of the model have to be represented in this logic. For each of the languages used in this chapter the first question is whether a shallow or a deep embedding of this language is to be used.

For a *deep embedding* the abstract syntax of the language is represented as a data type in the logic of the theorem prover and a semantic function interpreting all possible terms of the language is to be provided.

The advantage of a deep embedding are:

- Meta-theorems of the language can be formulated and verified with the theorem prover, because the semantics has already been formalised in the logic of the theorem prover.

[1] This is most of the time the better solution, because our experience suggest that in this case the specification contains an error.

- The translation into the language of the theorem prover is almost trivial and very easy to check, because only the abstract syntax tree (already present in the translator) has to be written in the logic of the theorem prover as an instance of the data type.

The disadvantage of a deep embedding are:

- Depending on the complexity of the semantics, reasoning can be very complex. Many properties of the language, which are not expressible in the type system of the theorem-prover's logic have to be proved for each specification, for example, type checking of the embedded program.

- If a language like OCL is to be embedded, part of the derivation rules and decision procedures of the theorem prover may have to be duplicated in the logic of the theorem prover as a theory, in order to be applicable to the embedded specification.

In case of *shallow embedding* the language is directly represented in the logic of the theorem prover. This requires translating the language into a *verification condition* encoding the semantics of the model. If this verification condition implies the specification, then the model is assumed to satisfy the specification.

The advantages of shallow embedding are:

- It is much simpler to manipulate shallowly-embedded formulae in PVS and one can use the highly automated rules provided by the theorem prover.

The disadvantages of shallow embedding are:

- The generation of the verification conditions has to be verified. If the translation is sufficiently simple, such a proof can be established, but in other cases one has to trust the verification condition generator.

Our translation is based on a hybrid approach. UML state machines and class diagrams are embedded deeply into the theorem prover, whereas OCL is embedded shallowly.

The syntax of UML state machines is considerably simpler than its semantics. In order to trust the embedding of state machines we need to be able to prove some properties of the semantics required by the standard. Furthermore, instead of defining a single semantics for state machines, the standard defines a collection of semantics through *semantic variation points*. These variation points are made explicit in the deep embedding, where an instance of the semantics can be obtained by, for example, instantiating functions for selecting the way events are processed.

4.3 PVS Language

PVS [121] is a proof assistant, proof checker, and a language for specifications in classical higher-order logic. Higher-order logic is basically a typed lambda-calculus enriched with the two logical operators $(\forall x : T)P(x)$ describing universal quantification over elements of type T, and implication ($\Longrightarrow$). One base type, boolean, of the underlying type theory is singled out in order to define predicates. See, for example, Leivant [93] for further a more detailed description of higher-order logic. In this section we describe the essential part of the PVS specification language used by our translator.

As higher-order logic is a typed language, so is PVS. A type is either one of the elementary types predefined by PVS, like boolean or integer, or it is a user-defined uninterpreted type. For example T : TYPE declares a new uninterpreted type T in PVS. The declaration T : TYPE+ declares a uninterpreted type which contains at least one element. Interpreted types can be defined by enumerating their member, which are uninterpreted constants of this type, for example: T : TYPE = $\{a, b, c\}$.

From the base types new types may be constructed. The most important construction is a function. Predicates and sets are defined by their characteristic functions, for example, a set of elements of types T has the type setof : TYPE = $[T \rightarrow \text{bool}]$.

Of similar importance are *predicate subtypes*. Predicate subtypes allow the definition of a new type by constraining an existing type using a predicate. The syntax of the definition of a predicate subtype is: S : TYPE = $\{x : T \mid P(x)\}$. This definition defines the type S as the subtype of T such that all individuals of S satisfy the predicate P.

Other type declarations we use are structures, which are declared by T : TYPE = $[\# f_1 : T_1, \ldots, f_n : T_n \#]$, where, for $1 \leq i \leq n$, the f_i are *field names*, which are used later to select values from a structure value, and the T_i specify the type of each field. For example, if e is some PVS expression whose type is T, then the value of f_1 in e is characterised by the expression $e`f_1$.

PVS allows the user to define new constants using a similar syntax as used for defining types. For example, a new constant which is a predicate is defined by, $P(x : T)$: bool = $Q(x)$. Alternatively, the same predicate can be defined using lambda-abstractions, for example, $P : [T \rightarrow \text{bool}]$ = LAMBDA $(x : T) : Q(x)$. Constants need not be interpreted, as in the preceding examples. Uninterpreted constants, for example, arbitrary predicates on T, are introduced by $C : [T \rightarrow \text{bool}]$.

Any predicate P can be used as a type by writing it as (P), which represents the set of values satisfying the predicate. This is often used in PVS specifications: for example, injective?, which is true if and only if its argument is an injective function, is used as a type: (injective?$[T, T]$) is the type of *injective function from T into T*.

In PVS all definitions have to be either constants or total functions. This is guaranteed by the type system of PVS, where the domain and the range of functions can be precisely described. For each function application the type checker may generate TCCs which prove that the arguments are inside the domain of the function. For recursive function definitions proof obligations are generated to prove termination, which ensure

that the function is total. For example, OCL's iterate expression (see Section 2.3) over a set can be defined in PVS as the following expression explained below:

```
iterate(s : finite_set[S], a : T, f : [S, T → T]) : RECURSIVE T =
    IF empty?(s)
    THEN a
    ELSE LET x = choose(s) IN iterate(remove(x, s), f(x, a), f)
    ENDIF
  MEASURE s BY strict_subset?
```
(4.1)

If the set to iterate over is empty, then the function returns the accumulator variable a. Otherwise, an element x of the input set s is chosen using Hilbert's epsilon function [15], the iterate expression f is applied to the current accumulator value a and the chosen element x to produce a new accumulator value, and the iterate function is recursively invoked with the new accumulator value, the iterate expression f, and $s \setminus \{x\}$ (written in PVS as "remove(x, s)") as the new set to iterate over. The MEASURE definition states that the argument s has to be measured by the strict subset relation $\subset$ and is used to generate the following proof obligations:

- The values of s are strictly ordered by $\subset$ and $(2^S, \subset)$ is a well-founded set.[2]
- With each recursive step the value of s is decreasing, that is: $s \setminus \{x\} \subset s$.

These conditions together imply that *iterate* is well-defined for all finite sets.

PVS specifications are organised into parameterised theories that may contain assumptions, definitions, axioms, and theorems. Such a theory is declared in PVS as, for example:

```
statemachine[Attribute, Class, Loc, Ref, Signal: TYPE+]: THEORY
  BEGIN
    ...
  END statemachine
```

More information about PVS and its language can be found in [138] and in [124].

4.4 Running Example

We use the *Sieve of Eratosthenes* as a running example. It is modelled using the two classes *Generator* and *Sieve* (see Figure 4.1). 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 numbers in increasing order, see Figure 4.2. The association from Generator to Sieve is called *itsSieve*.

[2] In PVS a relation $<$ on a type T is well founded, if $\forall P : P \subseteq T \implies (\exists (y : y \in P) \implies (\exists y : y \in P \wedge (\forall x : x \in P \implies \neg x < y)))$.

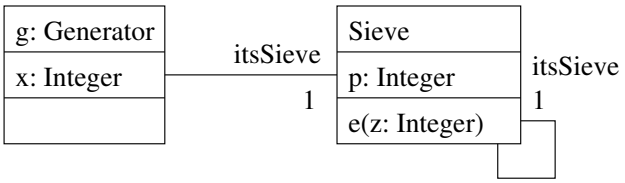

Figure 4.1: Class diagram of the Sieve example

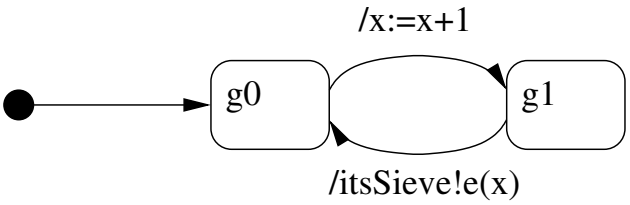

Figure 4.2: State machine of the Generator

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*, and starts receiving a sequence of integers i. If p divides i, then this instance does nothing. Otherwise, it sends i to *itsSieve*. This behaviour is shown in Figure 4.3.

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

$$\textbf{context } \mathit{Sieve} \textbf{ inv} : \mathit{Integer}\{2..(p-1)\} \rightarrow \mathit{forAll}(i \mid p.\mathit{mod}(i) \neq 0)$$

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 unbounded object creation and asynchronous communication, and therefore

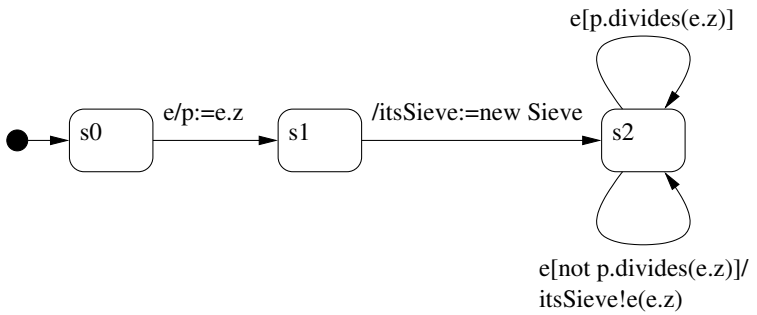

Figure 4.3: State machine of a Sieve

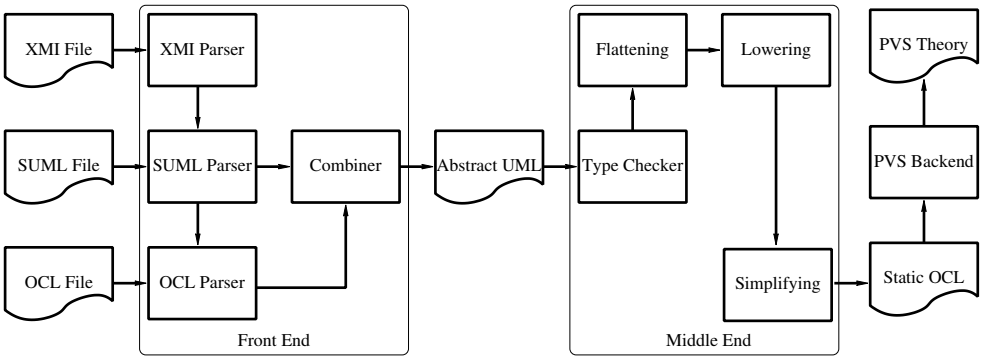

Figure 4.4: Architecture of the translator

does not have a finite state space. Furthermore, the behaviour of the model depends on the data sent between objects. Note also that the property we want to prove about 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.

4.5 Definition of the Translator

We define the translator from a subset of UML defined in this thesis into the input language of the theorem prover PVS. The restriction is, that we only allow associations in which an object is associated to at most one other object under the same association end name. Consequently, we may assume that Lemma 2.12 holds in this chapter.

The architecture of the translator is similar to that of a modern optimising compiler, using a front end for parsing input files, a middle end for analysing and transforming the parsed program into an equivalent, but optimised one, and a back end, which transforms the optimised tree into the target language (see the book of Aho, Sethi, and Ullman on compiler construction for details [4]). Here we describe the analysis and the transformations necessary for translating a UML model and its OCL requirements into the PVS language.

Figure 4.4 summarises the architecture of the translator and describes the data flow between the different parts of the system. The following sections explain each part of the translator in detail.

4.5.1 Front End

The front end is responsible for parsing the compilation units. The compilation units consists of files containing OCL constraints and the UML model.

We use existing modelling tools for creating the UML models. These tools are supposed to export the model using the Extensible Meta-data Interchange format XMI, a format specified by the Object Management Group for the exchange of UML models. This XMI format is available in five versions [103, 104, 106, 109, 108]; all of them use XML [157] to encode the model (and its abstract) syntax. All of these formats are sufficiently different, such that a different parser for each version of XMI is needed.

Another difficulty of the XMI format is that the specifications define an algorithm for generating an interchange format for a meta-model[3]. The UML standard defines a meta-model for UML models which most tools support for their diagram exchange; the generated XML document is very complex and contains a lot of redundancy. Because the UML standards are ambiguous, each tool vendor implements its own variant of the language and claims to implement XMI correctly.

To avoid implementing a parser parsing all versions and variants of XMI, which is a major engineering task, we have implemented translations from some supported variants to a language we call SUML (short for Simple UML format) using XSLT [50]. XSLT is a domain-specific language for transforming XML documents into other formats, especially other XML documents.

A SUML document, which is also an XML file, describes all concepts necessary for the tool's operation using a precisely specified abstract syntax. Parsing SUML is considerably simpler than parsing XMI because this format was designed to express a concept in precisely *one* way.

The SUML parser identifies all constraints present in an XMI or SUML file (unfortunately, many tools do not support embedding OCL into a model, and therefore do not store constraints in XMI files), and passes these to the OCL parser.

The syntax of OCL constraints supported by the parser is specified in [113]. A subset of the language is also described in Chapter 2.3.2. The OCL parser is used to parse constraints from a separate OCL file and from an SUML document into abstract syntax trees. The tool uses a recursive-descent parser, because OCL's grammar is in $LL(2)$, that is, it can be parsed top-down from left-to-right with 2 tokens of look-ahead and the parser creates a leftmost derivation.

Furthermore, the OCL grammar has been extended to parse a very simple action language, that is, a language used to specify the actions a state machine performs, which uses (a subset of) OCL as its expression language, and adds statements for signal emission, operation calls, conditional actions, loops, and assignments to attributes. These actions can be used in models to define the body of methods and to define the actions performed in state machines.

The abstract syntax trees generated by the SUML parser (representing class diagrams and state machines) and by the OCL parser (representing constraints and other expressions) are then passed to the combiner, which attaches each constraint to its context. Furthermore, it reports all constraints which do not have a context defined by

[3]The meta-model is an object-oriented implementation of an abstract syntax. See Section 1.1.

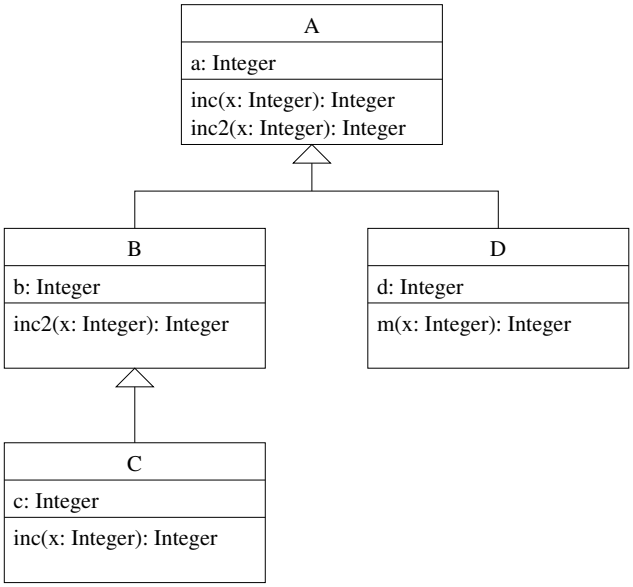

Figure 4.5: Example class diagram

a class diagram or a state machine. The result produced by the combiner is a single abstract syntax tree representing one UML model, which is passed to the middle end.

4.5.2 Middle End

The middle end is responsible for semantic analysis and semantics preserving transformations of the abstract syntax tree generated by the front end. This means, that all operations and transformations performed by the middle end are transformations of abstract syntax trees. The goal of these transformations are to transform the UML model into a form where it can be represented as a set of theories in PVS. The middle end is implemented in four phases: semantic analysis, flattening, lowering, and simplifying.

As a second running example, we use the model which consists of the class diagram in Figure 4.5.

Semantic Analysis

The first phase of the middle end consists of semantic analysis, and also contains the type checker. In this phase, the middle-end checks, whether the input model satisfies (a subset of) the well-formedness constraints specified in the standard, in particular, whether all constraints are well typed (as described in Chapter 3), annotates each use of an element (attribute, operation, and classes) in the syntax tree with its definition, whether all state machines are well-formed, and whether the class diagrams are well

formed (as described in Section 2.1). Note that this phase does not change the model or its semantics, it only annotates the model with information useful during later phases.

Furthermore, the translation may stop with reporting errors if the validation of the model fails. If validation fails, the tool has detected an inconsistency in the model, for example, a generalisation relation which is not a partial order, or one of the constraints is not well-typed.

One of these errors is the occurrence of a call to the function *oclIsUndefined*(). We have described the semantic issues with this function in Section 2.3.5, where we have explained that it is impossible to implement such a function. The behavior of this function, however, could be modeled in PVS. But this implies that the function eval has to be defined in PVS, which amounts to a deep embedding of OCL into PVS. Therefore, we have decided to remove the function from our subset of OCL.

Flattening

The second step is to flatten the model, that is, remove all uses of late binding from the model. Each call to a method is translated to a dynamic dispatch table, where the runtime-type of the callee is queried and the call of the method is statically resolved to a function. This step is necessary because PVS, as most other theorem provers, does not support late binding. Since the compiler is generating verification conditions, this transformation has to be performed by it.

Generating these calls also requires knowledge of the complete class diagram, because we need to know the overriding definitions of each operation called. If only part of the class diagram is used in this phase, then the compiler might miss an overriding definition of a method which this transformation does not take into account. The flattened tree would not call the overridden method, as required by the semantics, but one of the methods in a super-class.

For example, consider the class diagram displayed in Figure 4.5 and the OCL expression e.inc(2). The value returned by the call to inc depends on the runtime type of e, because we have a definition of this operation in classes A and C. By flattening, this dependency is made explicit by replacing the expression e.inc(2) with guarded static calls:

$$\begin{array}{l}\textbf{if } e.oclIsTypeOf(A) \textbf{ then } e.A :: \text{inc}(2)\\ \quad \textbf{else if } e.oclIsTypeOf(B) \textbf{ then } e.A :: \text{inc}(2)\\ \quad\quad \textbf{else } e.C :: \text{inc}(2) \textbf{ endif endif}\end{array}$$

Here, the notation $C :: \text{inc}(2)$ refers to a static call of inc(2) in class C and $A :: \text{inc}(2)$ to a static call of inc(2) in class A. Finally, if e is an instance of class B, then the implementation defined in class A is used, because objects of type B inherit the implementation in class A.

Lowering

The third step is to rewrite the abstract syntax tree, into one using simpler concepts. The most important part of lowering is to replace each occurrence of an, now statically determined, operation call by simpler expressions, provided that the meaning of an operation is defined using OCL.

An OCL definition can be obtained from a behavioural specification of an operation, if it is either specified using the pattern:

$$\textbf{context}\ c :: m(\vec{v}) : T\ \textbf{pre} : \text{true}\ \textbf{post} : \mathit{result} = e$$

or, more directly, using the declaration:

$$\textbf{context}\ c :: m(\vec{v}) : T\ \textbf{body} : e \quad .$$

The first pattern is quite common in the UML and in the OCL standard. If such a pattern is recognised, then the call to this operation is replaced by the expression e, where each formal parameter, which also occur in e, is replaced by the expression describing the actual arguments. This replacement is implemented as follows:

If we encounter a call $c :: m(\vec{e})$ in an OCL expression e' and we identify a definition e of $c :: m$, the call is replaced by

$$\textbf{let}\ f(\vec{v}) : T\ = e\ \textbf{in}\ f(\vec{e}) \quad ,$$

where f and $\vec{v}$ do not occur in e'. To handle the case of a recursive operation call, we replace calls to $c :: m$ in e by f, too. Observe that we have to re-introduce recursive let-definitions into OCL 2.0 [113], which were present in OCL 1.4 [105] but have been removed for the latest version of the standard.

Note that, after flattening, all calls are statically determined, so this transformation can almost always be performed. The only exception are mutually recursive definitions. In principle, mutually recursive functions can be obtained using the same mechanism in OCL, but the PVS specification language does not allow them.

Simplifying

Lowering usually introduces a lot of redundancy into the model, for example, useless cases in the dynamic dispatch tables and duplication of expressions. The simplifying pass uses algebraic laws to simplify all expressions. Therefore, the next pass, simplifying, is used to rewrite the model using algebraic simplifications.

The important part of this step is to reduce the number of cases which have to be distinguished for each call of an operation, whose semantics depends on the type of the callee.

4.5.3 Back End

The back end transforms an abstract syntax tree, which has been generated by the middle end, into the PVS specification language. The resulting theory is suitable as input for the PVS theorem prover.

One obvious shortcoming of the back end is that it cannot provide measure functions used in the type checker of PVS to verify that each function is computable and well-defined. The user may review the generated theory and provide these measures by hand-coding them into the PVS specification.

Class Diagrams

A class diagram is embedded deeply into PVS. In PVS we define a type Class which enumerates all classes occurring in the model. For example, the class diagram shown in Figure 4.1 leads to the definition:

$$\text{Class} : \text{TYPE+} = \{\text{OclAny}, \text{OclVoid}, \text{OclInvalid}, \text{Generator}, \text{Sieve}\} \quad .$$

Observe that OclAny, OclVoid, and OclInvalid are implicitly defined in the class diagram, as specified by OCL (see Section 2.3.4).

Next we define a subtype Active, which enumerates all classes occurring in Class which are active classes. In the sieve example, the classes Generator and Sieve are active, therefore, we define:

$$\text{Active} : \text{TYPE+} = \{x : \text{Class} \mid x = \text{Generator} \vee x = \text{Sieve}\} \quad .$$

The constant rootClass represents an active class from which the first object of the system is instantiated.

$$\text{rootClass} : \text{Active} = \text{Generator}$$

Furthermore, the attribute, operation, signal, and reference names of each class are enumerated as a type. Each name is prefixed with the name of the class which defines it.

Attribute : TYPE = {Generator___x, Sieve___z, Sieve___p, unusedAttribute}
Reference : TYPE = {Sieve___itsSieve, Generator___itsSieve,
Sieve___itsGenerator, unusedReference}
Operation : TYPE = {}
Signal : TYPE = {Sieve___e}

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 4.1 is presented in Figure 4.6.

As described in Section 2.2, objectstructures are used to interpret a class diagram. For sake of simplicity, we consider only associations where to each object at most one

```
Class: TYPE+ = { Generator, Sieve }
Active: TYPE+ = { x: Class | x=Generator OR x=Sieve }
rootClass: (active) = Generator
Attribute: TYPE+ = { Generator___x, Sieve___z, Sieve___p, unusedAttribute }
Reference: TYPE+ = { Sieve___itsSieve, Generator___itsSieve,Sieve___itsGenerator,
  unusedReference }
```

Figure 4.6: Translation of the Sieve class diagram

object can be associated. In this case, association end names can be treated as attributes of objects.

$$
\begin{array}{l}
\text{ObjectId} : \text{TYPE+} = \text{nat} \\
\textit{null} : \text{ObjectId} = 0 \\
\text{ObjectIdNotNull} : \text{TYPE+} = \{x : \text{ObjectId} \mid x \neq \textit{null}\} \\
\text{Object} : \text{TYPE} = [\#\text{class} : \text{Class}, \\
\qquad \text{aval} : [\text{Attribute} \rightarrow \text{Value}], \\
\qquad \text{rval} : [\text{Reference} \rightarrow \text{ObjectId}]\ \#] \\
\text{State} : \text{TYPE} = [\text{ObjectIdNotNull} \rightarrow \text{Object}]
\end{array}
\tag{4.2}
$$

ObjectId is the type of all object identifiers. We define *null* as an element of type ObjectId and define a subtype ObjectIdNotNull of ObjectId.[4] The type Object consists of a valuation function aval that assigns values to all attribute names, a function rval that assigns values to all reference names (or association end names), and a field class referencing the class of which the object is an instance of. Finally, a (global) state, or object diagram is defined as a function from ObjectIdNotNull to Object. Observe, that *null* does not have an object in this state. Also, whenever an object is accessed, PVS will generate type consistency constraints which require the user to prove that the object accessed is not *null*.

Type checking in the middle-end asserts that all objects only uses attributes defined for their class, so we assign to the attributes not defined in an object's class an arbitrary value. Therefore, we have deliberately simplified the back-end to generate for each object the attributes occurring in all classes of the model. A value for this attribute is specified only if the attribute occurs in the full descriptor of the class of the object.

Because PVS only allows total functions, we have an infinite number of objects in the state (it is indeed defined as a function nat → Object). In the semantics, we assign to each "slot" in the state, which represents an object which does not exists, an "instance" of OclInvalid. Using the natural numbers, we can obtain a slot for a new object by the PVS expression new(s : State) : ObjectIdNotNull = min{a : ObjectIdNotNull | class(state(a)) = OclInvalid}.

[4] Using natural numbers as object identifiers (indicated by the PVS type nat) is only a convenience. We could have left the type ObjectId uninterpreted.

State Machines

We use a hybrid approach for embedding state machines into PVS, using a deep embedding to represent the structure of the state machine and a shallow embedding of guards and expressions.

The back-end generates a type for all the locations of the state machines occurring in the model, prefixing it with the name of the class in which the state has been defined. Because states need not be named in UML and need not be unique for a state machine, we use an arbitrary number or string, which is defined in the XMI file. Therefore, the locations of the sieve example are:

$$\begin{aligned}\text{Location} : \text{TYPE} = \{&\text{Generator\_\_\_\_61}, \text{Generator\_\_\_\_64}, \text{Generator\_\_\_\_66},\\ &\text{Sieve\_\_\_\_25}, \text{Sieve\_\_\_\_28}, \text{Sieve\_\_\_\_32}, \text{Sieve\_\_\_\_33}\}\end{aligned}$$

Similar to the way active classes are represented, we define a subtype of Location containing all initial locations:

$$\text{Initial} : \text{TYPE} = \{\ell : \text{Location} \mid \ell = \text{Sieve\_\_\_\_25} \vee \ell = \text{Generator\_\_\_\_61}\}$$

Transitions are embedded deeply, but parts of it use a shallow embedding. A transition is defined by the class of the state machine, a source location, a target location, a trigger, a guard, which is any PVS function mapping a global state into the booleans, and a list of actions defined in the action language. The type of a transition is:

$$\begin{aligned}\text{Transition} : \text{TYPE} = [&\#\text{class} : \text{Class}, \text{source} : \text{Location}, \text{target} : \text{Location},\\ &\text{trigger} : \text{Event}, \text{guard} : [\text{State} \to \text{bool}], \text{action} : \text{List[Action]}, \#]\end{aligned}$$

Next, we define all transitions occurring in the state machine as constants of type Transition. Again, the names are chosen using the XMI representation, such that they are unique for the complete PVS theory. For example, the transition from *state_1* to *state_1* sending an integer to the next object, as shown in Figure 4.2, is translated to:

$$\begin{aligned}&t28 : \text{Transition} = (\#\text{source} := \text{Sieve\_\_\_\_28},\\ &\quad \text{trigger} := \text{signalEvent}(\text{Sieve\_\_\_}e, \text{Sieve\_\_\_}z),\\ &\quad \text{guard} := \lambda(\text{val} : \text{State}) : (\neg\text{divides}((\text{val`aval}(\text{Sieve\_\_\_}p)),\\ &\qquad (\text{val`aval}(\text{Sieve\_\_\_}z)))),\\ &\quad \text{actions} := (\text{cons}((\text{emitSignal}(\text{Sieve\_\_\_itsSieve}, \text{Sieve\_\_\_}e,\\ &\qquad \lambda(\text{val} : \text{State}) : (\text{val`aval}(\text{Sieve\_\_\_}z))))), \text{null}),\\ &\quad \text{target} := \text{Sieve\_\_\_\_28},\\ &\quad \text{class} := \text{Sieve}\ \#)\end{aligned}$$

Finally, the translator defines a set of all transitions occurring in the model. The result of this enumeration of transitions is shown in Figure 4.7.

The semantics of the translated state machines is given in terms of sets of computations using a PVS theory provided by Hooman and van der Zwaag and described in [67].

```
Location: TYPE+ = { Generator____61, Generator____64, Generator____66,
   Sieve____25, Sieve____28, Sieve____32, Sieve____33 }

Initial:TYPE={l: Location | l = Sieve____25 OR l = Generator____61 }

⋮

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 }
```

Figure 4.7: Translation of the Generator state machine

OCL

Finally, we have to generate a specification for OCL expressions and actions. This involves "embedding" the three-valued logic of OCL (see Section 2.3.5) into the two-valued logic of PVS .

For this embedding, we use the fact that the OCL standard requires that all constraints have to be true for a well-formed model. Because we are interested in verifying the *correctness* of a model with respect to its specification, we actually have to prove that all OCL constraints are *true*, that is, neither *false* nor *undefined*. If an expression evaluates to undefined, then the constraint containing it should not be provable.

We do this by using a shallow embedding, such that all constraints that evaluate to true also evaluate to true in PVS and all constraints which evaluate to false also evaluate to false in PVS. The translation was defined in a way that constraints which evaluate to undefined ($\mathcal{J}$) using the evaluation function defined in Definition 2.19 lead to unprovable type consistency constraints. Consequently, models containing constraints with subexpressions that evaluate to undefined are not provable in PVS.

1. We do not need to require that each function is strict in its arguments. Instead, we use the requirement that each function application yields a well-defined value. PVS implements this requirement by generating type consistency constraints automatically, in order to require a proof that each argument supplied is in the domain of the function and that recursively defined functions always have a defined value for each of its arguments.

2. We do not have to redefine the core logic, for example, the *and* and *implies* functions, to properly define the functions described in Table 2.2, in PVS. This entails the development of specialised strategies to automate reasoning in a three-valued logic. Using our approach we benefit from PVS's high degree of automation.

3. We removed the function *oclIsUndefined* from OCL, that is, the tool does not translate models which contain constraints using this function. For this function we were not able to give a definition without implementing a deep embedding and we want to avoid a deep embedding. This requires that the user rewrites the constraints in such a way that he does not need to use *oclIsUndefined*. This can be done by identifying the condition φ under which constraint ψ may be undefined and writing a corresponding constraint $\neg\varphi \wedge \psi$, if one expects the constraint to be false if it is undefined, or $\neg\varphi \implies \psi$ if he does not care about these undefined cases.

Primitive functions. Some primitive functions used in OCL are partial functions which may result in undefined, for example, division if the dividend is zero. In [20] Brucker and Wolff have extended the partial functions to total functions by explicitly introducing undefined and formalising the underlying three-valued logic. This

approach is a deep embedding, which enabled them to prove certain meta-theoretic properties about different embeddings in OCL. The main drawback for verifying a concrete system is that by reasoning in a three-valued logic one loses PVS's high degree of automation. The main reason is, that for OCL the law of the excluded middle ($\vdash p \vee \neg p$) and the axiom $\vdash p \implies p$ do not hold. However, in PVS it is assumed that these axioms always hold in its strategies.

Instead, we *restrict* each partial function to its domain making it a total function. This requires formalising the domains of each primitive function defined in OCL. Most functions of the OCL standard library already have an equivalent definition in PVS, for example, the arithmetic functions. We have defined a library of total functions which are not provided by PVS.

A consequence of this encoding is, that whenever an expression in OCL may evaluate to undefined, the corresponding expression in PVS has an unprovable type consistency constraint. Conversely, if all type consistency constraints of the translated PVS expression are provable, the original OCL expression never evaluates to undefined!

Null references. 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. Such functions have to be represented in PVS. We can define such a definition if the meaning of the function is given as an OCL expression or if its semantics has been defined using PVS expressions.

The signature of such a function is obtained from the type definitions computed from the class hierarchy. All instances in a model are identified by a value of the type *OclAny*, which is represented in PVS by the type ObjectId. This type contains a special object *null* which represents the non-existing object. We refine these types to reflect the class hierarchy using predicate subtypes. First the back-end generates a partial order $\preceq$ which encodes the generalisation hierarchy of the class diagram. For the Sieve example, this predicate is:

$$\preceq : (\text{partial\_order?}[\text{Class}]) = \lambda(x, y : \text{Class}) : (x = y) \vee (x = \text{Generator} \wedge y = \text{OclAny}) \vee (x = \text{Sieve} \wedge y = \text{OclAny}) \quad .$$

Then for each class C defined in the class diagram a predicate subtype

$$C\text{IdNotNull} = \{x : \text{ObjectIdNotNull} \mid x\text{`class} \preceq C\} \quad .$$

is defined. This encodes the usual the subsumption property that if a class C is a subclass of D, then each instance of C is also an instance of D. Because parameters may be *null*, we define

$$C\text{Id} = \{x : \text{ObjectId} \mid x = \mathit{null} \vee x\text{`class} \preceq C\} \quad .$$

The signature of a user-defined method $C :: m(v_1 : T_1, v_2 : T_2, \ldots, v_n : T_n) : T$ will be defined as:

$$C\_\_\_m : [C\text{IdNotNull}, T_1\text{Id}, T_2\text{Id}, \ldots, T_n\text{Id} \rightarrow T\text{Id}]$$

and a corresponding function encoding the semantics of the OCL function as defined below.

Undefined values, which occur, for example, by accessing attributes of *null* or array members outside of the bounds of the array, or retyping (downward-casting) an object to a subtype of its real type, are avoided by the formalisation of signatures. For example, requiring that the first parameter of a PVS function, which is used for the *self* reference, is non-null causes the proof checker to generate a TCC, which establishes that the value of *self* is never *Null*. Furthermore, type checking guarantees that for every object the value of each of its attributes is defined. This property is maintained by the behavioural semantics.

Recursion. The formal semantics of OCL is concerned with *executing* OCL constraints. Consequently, the value of an recursive function is undefined, if its evaluation is diverging. These problems do not occur in PVS, because it is not possible to define a diverging recursive function in PVS. For each recursive function definition a ranking function has to be supplied, which is used in a termination proof. We translate recursive functions of OCL to recursive functions in PVS directly and ask the user to supply a suitable ranking function in the PVS output, because it is not possible to automatically compute such a function.

Example 4.1. In OCL one might define Fibonacci's function recursively as:

> **context** *Integer* :: fib() : *Integer*
> **pre** : *self* ≥ 0
> **body** : **if** *self* > 1 **then** (*self* $- 2$).fib() + (*self* $- 1$).fib() **else** 1 **endif**

Assuming that *Integer* is not sub-classed, this expression is translated to PVS as:

```
Integer___fib(self : int | x ≥ 0) : RECURSIVE int =
    IF self > 1 THEN Integer___fib(self − 2) + Integer___fib(self − 1) ELSE 1 ENDIF
  MEASURE ...
```

The measure function is not generated, because it cannot be generally computed automatically. ♦

As an alternative we propose to add a ranking function on the level of OCL, for example, using a stereotype **rank** mapping the arguments of the recursive functions into the non-negative integers.

Example 4.2. Extending the definition of Fibonacci's function with a rank we obtain:

> **context** *Integer* :: fib() : *Integer*
> **pre** : *self* ≥ 0
> **body** : **if** *self* > 1 **then** (*self* − 2).fib() + (*self* − 1).fib() **else** 1 **endif**
> **rank** : *self*

Now this expression is translated to PVS as:

```
Integer___fib(self : int | x ≥ 0) : RECURSIVE int =
    IF self > 1 THEN Integer___fib(self − 2) + Integer___fib(self − 1) ELSE 1 ENDIF
  MEASURE self
```

In this case a number of type consistency constrains are generated which establish that the expression is well-typed and that the function is totally defined. For example, one of the termination constraints, which establishes that the rank is decreasing, is: self > 1 ⟹ self − 2 < self. ♦

Definition of the Translation. Having given an overview of the central ideas of the translation, we formally define the translation function *trans* in this section and comment on each decision.

Definition 4.1 (Translation of Types)**.** The translation of types is defined by:

1. $trans(Boolean) \stackrel{\text{def}}{=} \text{bool}$
2. $trans(Integer) \stackrel{\text{def}}{=} \text{int}$
3. $trans(Real) \stackrel{\text{def}}{=} \text{real}$
4. $trans(Bag(T)) \stackrel{\text{def}}{=} \text{finite\_bag}[trans(T)]$
5. $trans(Set(T)) \stackrel{\text{def}}{=} \text{finite\_set}[trans(T)]$
6. $trans(Sequence(T)) \stackrel{\text{def}}{=} \text{finseq}[trans(T)]$
7. $trans(C) \stackrel{\text{def}}{=} C\text{Id}$ ◊

Definition 4.2 (Translation of Expressions)**.** Let $\mathcal{V}$ be a set of names we call *variables*. Let D be a class diagram. In the following, state and prestate are two free variables. The resulting PVS expression will then be bound by lambda abstractions to obtain a function on actions to booleans.

1. *trans*(true) = true, *trans*(false) = false, and *trans*(null) = *null*.
2. For each number n define $trans(n) = n$.

3. For collection expressions, we show the case of representing a sequence, the other collections are analogous: Let $e_1, e_2, \ldots, e_n$ be a list of OCL expressions, which may be empty. Then

$$trans(Sequence\{e_1, e_2, \ldots, e_n\}) = \text{cons}(trans(e_1), trans(Sequence\{e_2, \ldots, e_n\}))$$

and

$$trans(Sequence\{\}) = \text{null} \quad .$$

For a range expression $e..e'$ define $trans(e..e') = \text{range}(T(e), T(e'))$, where range is defined in a library as:

$$\begin{array}{l} \text{range}(x : \text{int}, y : \text{int}) : \textsc{recursive}\ \text{list}[int] = \\ \quad \textsc{if}\ x < y \\ \quad \textsc{then}\ \text{cons}(x, \text{range}(x + 1, y)) \\ \quad \textsc{else}\ \text{null} \\ \quad \textsc{endif} \\ \textsc{measure}\ y - x \end{array}$$

4. For any variable $v \in \mathcal{V}$ define $trans(v) = v$.

5. For conditional expressions define

$$trans(\textbf{if}\ t\ \textbf{then}\ t_1\ \textbf{else}\ t_2\ \textbf{endif}) = \textsc{if}\ trans(t)\ \textsc{then}\ trans(t_1)\ \textsc{else}\ trans(t_2)\ \textsc{endif} \quad .$$

6. For let-expressions define

$$\begin{array}{l} trans(\textbf{let}\ v_0(\vec{v_0}) : T_0 = t_0, \ldots, v_n(\vec{v_n}) : T_n = t_n\ \textbf{in}\ t) = \\ \quad \textsc{let}\ trans(v_0)(trans(\vec{v_0})) : trans(T_0) = trans(t_0), \\ \qquad \ldots \\ \qquad trans(v_n)(trans(\vec{v_n})) : trans(T_n) = trans(t_n) \\ \quad \textsc{in}\ trans(t) \quad . \end{array}$$

7. For attribute call expressions define

$$trans(t.a) = \text{state}(trans(t))\text{`aval}(\text{pvsattrname}(t, a)) \quad ,$$

where pvsattrname obtains the type of t deduced by the type checker, looks up the class in which a has been defined and returns the name defined for this attribute in the PVS formalisation of PVS, as described above. Recall, that the member aval of the structure of type Object refers to the valuation of an objects attribute.

8. For previous attribute call expressions define

$$trans(t.a@\mathbf{pre}) = \text{prestate}(trans(t))\text{`aval}(\text{pvsattrname}(t, a)) \quad .$$

9. For operation call expressions, where the type of t_0 is not a collection type, define

$$trans(t_0.m(t_1, t_2, \ldots, t_n)) = \\ \text{pvsopername}(t_0, m)(trans(t_0), trans(t_1), trans(t_2), \ldots, trans(t_n)) \quad ,$$

where pvsopername(t_0, m) expands to the name of a function defined in the PVS library or computed from the model, which implements the semantics of m in PVS.[5]

For operation call expressions, where the type of t_0 is of type sequence, define:

$$trans(t_0.m(t_1, t_2, \ldots, t_n)) = \text{map}(trans(t_0), \\ \text{LAMBDA } x : \text{pvsopername}(t_0, m)(x, trans(t_1), trans(t_2), \ldots, trans(t_n))) \quad .$$

The other collection types use analogous definitions.

10. For collection property call expressions define:

$$trans(t_0 \rightarrow m(t_1, t_2, \ldots, t_n)) = \\ \text{pvsopername}(t_0, m)(trans(t_0), trans(t_1), trans(t_2), \ldots, trans(t_n)) \quad ,$$

where pvsopername(t_0, m) expands to the name of a function defined in the PVS library defining the semantics of the call.

11. For an iterate expressions define

$$trans(t_0 \rightarrow iterate(v_0 : T_0; a : T = t \mid t_n)) = \\ \text{iterate}(trans(t_0), trans(t), \\ \text{LAMBDA } (v_0 : trans(T_0), a : trans(T)) : \ trans(t_n))$$

For quantification expressions, however, we define:

a) $trans(t_0 \rightarrow forAll(v_0 : T_0 \mid t_n)) = \text{FORALL } (v_0 : (trans(t_0))) : trans(t_n)$

b) $trans(t_0 \rightarrow exists(v_0 : T_0 \mid t_n)) = \text{EXISTS } (v_0 : (trans(t_0))) : trans(t_n)$

[5] The definition of pvsopername(t, m) for operations and types defined in the OCL standard library is described in Appendix B.

12. For a flatten expression define

$$trans(t \rightarrow flatten()) = \text{flatten}(trans(t)) \quad ,$$

if the type of t is a collection of collections, and, otherwise, define:

$$trans(t \rightarrow flatten()) = trans(t) \quad .$$

13. For all instances of T expressions define

$$trans(T.allInstances()) = \{x : \text{ObjectIdNotNull} \mid \text{state}(x)\text{`class} = trans(T)\}$$

if T is an object type, and

$$trans(T.allInstances()) = \{x : trans(T) \mid \textsc{True}\} \quad ,$$

otherwise. For the @**pre** modified expression, replace state with prestate in the above translations. ◊

Looking at the definition of the translation function, it becomes apparent that the generated PVS functions have a structure very similar to the original OCL constraint, while the generated PVS specification does not embed a three-valued logic. The advantage of this approach is that proofs of the verification conditions are easier to find.

4.6 Soundness of the Translation

We establish the soundness of our translation. By soundness we mean that for all OCL constraint φ, if their translation is provable, then it evaluates to true in any model. For the proofs in this section, we assume the following:

Assumption 4.3. The logic used by PVS and its implementation is sound.

The logic used by PVS is well-understood and sound (see Owre and Shankar [123] for references). Whether the implementation and the "oracles", that is, automatic decision procedures like model checkers, used by PVS are sound is unknown, because the PVS system has not yet been formally verified. Therefore we have to assume the soundness of the system.

The first step of the soundness proof is relating the carriers of types, described by a function $\mathfrak{D} : \mathcal{T} \rightarrow U_\omega$, in OCL, as defined in Definition 2.18, to their translated types in PVS, as obtained by Definition 4.1. The carriers of the translated types in PVS are obtained from the carriers of the original types in OCL by removing the values $\mathcal{J}$ from the carriers in OCL , as expressed by the following proposition:

Proposition 4.4. *For the translation of types the following properties hold:*

1. $\mathfrak{D}_{\mathrm{OCL}}(\mathit{Boolean}) \setminus \{\mathcal{J}_{\mathbb{B}}\} = \mathfrak{D}_{\mathrm{PVS}}(\mathrm{bool})$.
2. $\mathfrak{D}_{\mathrm{OCL}}(\mathit{Integer}) \setminus \{\mathcal{J}_{\mathbb{R}}\} = \mathfrak{D}_{\mathrm{PVS}}(\mathrm{int})$.
3. $\mathfrak{D}_{\mathrm{OCL}}(\mathit{Real}) \setminus \{\mathcal{J}_{\mathbb{R}}\} = \mathfrak{D}_{\mathrm{PVS}}(\mathrm{real})$.
4. *For all types* τ *it holds* $\mathfrak{D}_{\mathrm{OCL}}(\mathit{Bag}(\tau)) \setminus \{\mathcal{J}_{\mathbb{N}^{\mathfrak{D}(\tau)}}\} = \mathfrak{D}_{\mathrm{PVS}}(\mathrm{finite\_bag}[\mathit{trans}(\tau)])$.
5. *For all types* τ *it holds* $\mathfrak{D}_{\mathrm{OCL}}(\mathit{Set}(\tau)) \setminus \{\mathcal{J}_{2^{\mathfrak{D}(\tau)}}\} = \mathfrak{D}_{\mathrm{PVS}}(\mathrm{finite\_set}[\mathit{trans}(\tau)])$.
6. *For all types* τ *it holds* $\mathfrak{D}_{\mathrm{OCL}}(\mathit{Sequence}(\tau)) \setminus \{\mathcal{J}_{\mathfrak{D}(\tau)^{\mathbb{N}}}\} = \mathfrak{D}_{\mathrm{PVS}}(\mathrm{finseq}[\mathit{trans}(\tau)])$.
7. *For all object types* τ *it holds:* $\mathfrak{D}_{\mathrm{OCL}}(\tau) \setminus \{\mathcal{J}_{\mathbb{O}}\} = \mathfrak{D}_{\mathrm{PVS}}(\tau\mathrm{Id})$. *Recall that object types are of kind* $\star$.

Proof. The proof is by induction on the construction of types. □

From now on we do not distinguish the carriers of types of OCL from the ones of PVS, because they have the same semantics, only the names of the types differ.

Knowing that the carriers of the types agree in PVS and OCL we lift this property to the formalisation of object diagrams, as defined in Definition 2.10, in PVS, which is defined in Equation (4.2). This is expressed in the next lemma.

Lemma 4.5. *For all object diagrams* $\sigma \in \Sigma$ *there exists a* $v \in \mathfrak{D}(\mathrm{State})$ *such that there exists an isomorphism between* v *and* σ*, written* $v \simeq \sigma$.[6]

Proof. Let Σ be the set of all object diagrams and $\sigma = \langle \tau, \xi, [\![\cdot]\!] \rangle \in \Sigma$ an object diagram.

By definition of $\mathbb{O}$ as a countable enumerable set, we find a bijective function $f_{\mathbb{O}}$ between $\mathbb{O}$ and $\mathbb{N}$ such that $f_{\mathbb{O}}(\mathit{null}) = 0$.

Let o be an object. An object state is defined in PVS as: (# class := $\tau(o)$, aval := $\xi(o)$, rval := $\lambda e : \iota x : o \overset{e}{\twoheadrightarrow} x$ #) (recall, that in the setting of this chapter encoding associations as attributes named by the association end names is justified by Lemma 2.12). For each object identifier o which is not defined in τ define the object state in PVS by (# class := OclInvalid, aval := $\lambda n : 0$, rval := $\lambda n : \mathit{null}$ #). We say that an object state ω is of the same form as its object o in PVS, written $\omega \simeq o$, if the one can be obtained from the other using the above construction.

Finally, define the instance s of State by point-wise extension, such that for all $o \in \mathbb{O}$ we have $s(f_{\mathbb{O}}(o)) \simeq \sigma(o)$. □

Next, we consider the valuation of freely occurring local variables. Assume that the translation of a well-typed OCL formula is well-typed. Then the translation of variables in Definition 4.2, Item 4 indicates that we may use the local valuations η, which interpret the OCL constraint, for interpreting the translated formula in PVS, too.

[6] See Equation (4.2) for the definition of State.

In PVS we do not have a value for undefined. One only has a proof of a property if all TCCs generated for a theory have been proved. Only if these type consistency constraints have been proved, a theory in PVS is considered well-typed. Consequently, we have to prove, that if a term is well-typed in PVS, then the value of the corresponding OCL constraint is not undefined. This is established by the following lemma:

Lemma 4.6. *Let φ be a well-typed OCL constraint such that trans(φ) is well-typed in PVS, which implies that all type consistency constraints are provable. Then for all $\overleftarrow{\sigma}, \sigma$ and valuation η of the local variables the equation* $\text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(\varphi) = [\![\mathit{trans}(\varphi)]\!]\eta$ *holds.*

Proof. Let φ be an OCL constraint. The proof is by structural induction on the construction of φ, showing that for each application of $\text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)$ there is a corresponding step in PVS.

Let $\overleftarrow{s}$ and s be states in PVS such that $\overleftarrow{s} \simeq \overleftarrow{\sigma}$ and $s \simeq \sigma$, where $\simeq$ is defined as in the proof of Lemma 4.5.

1. If $\mathit{trans}(\varphi) = \textsc{true}$, then $\varphi = \text{true}$, and consequently $\text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(\text{true}) = [\![\textsc{true}]\!] = \text{true}$.

   If $\mathit{trans}(\varphi) = \textsc{false}$, then $\varphi = \text{false}$, and consequently $\text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(\text{false}) = [\![\textsc{false}]\!] = \text{false}$.

   If $\mathit{trans}(\varphi) = \textsc{null}$, then $\varphi = \text{null}$, and, as a consequence, $\text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(\text{null}) = [\![\textsc{null}]\!] = 0$.

2. For numeric literals ℓ it holds that $\mathit{trans}(\ell) = \ell$, and, as a consequence, in all environments η $\text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(\ell) = [\![\ell]\!]$ holds.

3. Let $e_1, e_2, \ldots, e_n$ be a list of OCL expressions, which may be empty (in which case $n = 0$), such that $\text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(e_i) = [\![\mathit{trans}(e_i)]\!]\eta$ holds for all $1 \leq i \leq n$ as an induction hypothesis. We prove the case of $\mathit{Sequence}\{e_1, e_2, \ldots, e_n\}$ by induction on i.

   For empty sequences, which satisfy $i = 0$, $\mathit{trans}(\mathit{Sequence}\{\}) = \mathit{null}$ holds and both $\text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(\mathit{Sequence}\{\})$ and $[\![\mathit{null}]\!]$ result in the empty sequence.

   For singleton sequences, which satisfy $i = 1$,

   $$\mathit{trans}(\mathit{Sequence}\{e_1\}) = \text{cons}(\mathit{trans}(e_1), \mathit{null})$$

   holds and, by induction hypothesis $\text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(e_1) = [\![\mathit{trans}(e_1)]\!]\eta$, we conclude that

   $$\text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(\mathit{Sequence}\{e_1\}) = [\![\text{cons}(\mathit{trans}(e_1), \mathit{null})]\!]\eta \quad .$$

Finally, assume $i > 1$ and, as additional induction hypothesis:

$$\text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(\mathit{Sequence}\{e_1, e_2, \ldots, e_{i-1}\}) = \\ [\![\mathit{trans}(\mathit{Sequence}\{e_1, e_2, \ldots, e_{i-1}\})]\!]\eta \quad . \tag{4.3}$$

By observing that

$$\text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(\mathit{Sequence}\{e_1, e_2, \ldots, e_i\}) = \\ \text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(\mathit{Sequence}\{e_1, e_2, \ldots, e_{i-1}\} \rightarrow \mathit{append}(e_i))$$

and by using the semantics of the *append* method, we conclude

$$\text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(\mathit{Sequence}\{e_1, e_2, \ldots, e_i\}) = \\ \text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(\mathit{Sequence}\{e_1, e_2, \ldots, e_{i-1}\}) \cdot \text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(e_i) \quad .$$

Next, we use the fact, that cons describes $\cdot$ in PVS, such that

$$[\![\mathit{trans}(\mathit{Sequence}\{e_1, e_2, \ldots, e_i\})]\!]\eta = \\ [\![\mathit{trans}(\mathit{Sequence}\{e_1, e_2, \ldots, e_{i-1}\} \rightarrow \mathit{append}(e_i))]\!]\eta$$

Using the induction hypothesis (4.3) and $\text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(e_i) = [\![\mathit{trans}(e_i)]\!]\eta$ we conclude that for all $0 \leq i \leq n$ we have

$$\text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(\mathit{Sequence}\{e_1, e_2, \ldots, e_i\}) = [\![\mathit{trans}(\mathit{Sequence}\{e_1, e_2, \ldots, e_i\})]\!]\eta \quad ,$$

which, for the case $i = n$ establishes the claim for this case.

It remains, that $\text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(e..e') = [\![\mathit{trans}(e..e')]\!]\eta$ holds for range expressions $e..e'$. This follows the proof of this case applied to Definition 2.19, Item 3c, and Definition 4.2.

4. If v is a variable expression, then $\mathit{trans}(v) = v$, and, as a consequence:

$$\text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(v) = \eta(v) = [\![v]\!]\eta \quad .$$

5. Let t', t'', t''' be OCL expressions and $t =$ **if** t' **then** t'' **else** t'''**endif**.

   By Definition 2.19, Item 5 we have:

$$\mathit{trans}(t) = \text{IF } \mathit{trans}(t') \text{ THEN } \mathit{trans}(t'') \text{ ELSE } \mathit{trans}(t''') \text{ ENDIF} \quad .$$

   Then $\mathit{trans}(t)$ is, by assumption, well typed, $\mathit{trans}(t')$ has type bool and $\mathit{trans}(t'')$ and $\mathit{trans}(t''')$ are expressions of the same type in PVS, say τ.

Because $trans(t)$ is well-typed in PVS, we may conclude from the induction hypothesis:

$$\text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(t') = [\![trans(t')]\!]\eta \quad .$$

If $[\![trans(t')]\!]\eta = \text{true}$, then $[\![trans(t)]\!]\eta = [\![trans(t'')]\!]\eta$ by the semantics of PVS. From the induction hypothesis we then derive:

$$\text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(t'') = [\![trans(t'')]\!]\eta \quad .$$

If $[\![trans(t')]\!]\eta = \text{false}$, then $[\![trans(t)]\!]\eta = [\![trans(t''')]\!]\eta$ by the semantics of PVS. From the induction hypothesis we then derive:

$$\text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(t''') = [\![trans(t''')]\!]\eta \quad .$$

Note, that because $trans(t')$ is well-typed in PVS, we may conclude from the induction hypothesis: $\text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(t') \neq \mathcal{J}_{\mathbb{B}}$.

Therefore, in any case we conclude:

$$\text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(\textbf{if } t' \textbf{ then } t'' \textbf{ else } t''' \textbf{ endif}) = [\![\text{IF } trans(t') \text{ THEN } trans(t'') \text{ ELSE } trans(t''') \text{ ENDIF}]\!]\eta \quad .$$

6. Let $t, t_0, \ldots, t_n$ are OCL expressions, $v_0, \ldots, v_n$ variable names, and $\vec{v}_0, \ldots, \vec{v}_n$ lists of variable declarations (of the form $v : T$, where v is a variable name and T is a type), and $T_0, \ldots, T_n$ type names. By observing that the OCL evaluation function implements an environment-based, applicative order, and left-to-right evaluating abstract machine for a typed lambda, and the fact that **let** and LET have the same reduction semantics, we can conclude:

$$\begin{aligned}[\![trans(\textbf{let } v_0(\vec{v}_0) : T_0 = t_0, \ldots, v_n(\vec{v}_n) : T_n = t_n \textbf{ in } t)]\!]\eta &= \\ [\![\text{LET } v_0(\vec{v}_0) : T_0 = t_0, \ldots, v_n(\vec{v}_n) : T_n = t_n \text{ IN } t]\!]\eta &= \\ \text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(\textbf{let } v_0(\vec{v}_0) : T_0 = t_0, \ldots, v_n(\vec{v}_n) : T_n = t_n \textbf{ in } t) \quad .\end{aligned}$$

7. a) Let a be an attribute name, t be an OCL expression, and, as an induction hypothesis, we assume $\text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(t) = [\![trans(t)]\!]\eta$. By assumption, $trans(t.a)$ is well typed in PVS, which implies $[\![trans(t)]\!]\eta \neq null$. As a consequence $[\![trans(t.a)]\!]\eta$ is not undefined. Then we have by Definition 2.19:

$$\text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(t.a) = \xi(\text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(t))(a)$$

and by Definition 4.2 we have:

$$trans(t.a) = \text{state}(trans(t))\text{`aval}(\text{pvsattrname}(t, a)) \quad .$$

Using Lemma 4.5 we conclude:

$$\text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(t.a) = [\![trans(t.a)]\!]\eta \quad .$$

b) For navigation expressions $t.e$, where t evaluates to an object identity, assume as induction hypothesis, that $\text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(t) = [\![trans(t)]\!]\eta$. Then we have by Definition 2.19 and using Lemma 2.12:

$$\text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(t.e) = \iota x : (\text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(t)) \overset{e}{\rightarrow} x$$

and by Definition 4.2 we have:

$$trans(t.e) = \text{state}(trans(t))\text{`rval}(\text{pvsattrname}(t, e)) \quad .$$

Using Lemma 4.5 we conclude:

$$\text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(t.e) = [\![trans(t.e)]\!]\eta \quad .$$

8. For expressions of the form $t.v@\textbf{pre}$ the proof is similar to Case 7.

9. Let m be an operation name and $t_0, t_1, \ldots, t_n$ be a sequence of OCL expressions such that, as induction hypothesis, we have $\text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(t_i) = [\![trans(t_i)]\!]\eta$ for all $0 \leq i \leq n$. Furthermore, assume that $trans(t_0.m(t_1, \ldots, t_n)$ is well-typed in PVS. This implies, that $[\![trans(t_0)]\!]\eta \neq null$. Furthermore, because all type-checking constraints are provable, which implies that in PVS the expression $trans(m)(trans(t_0), trans(t_1), \ldots, trans(t_n))$ is not undefined for all interpretations of m. Consequently:

$$\text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(t_0.m(t_1, \ldots, t_n) \neq \mathcal{J}$$

for any undefined value $\mathcal{J}$.

Note, that proving $\text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(t_0.m(t_1, \ldots, t_n)) = [\![trans(t_0.m(t_1, \ldots, t_n))]\!]\eta$ has to be done for each operation defined in the model and its translation in PVS individually. The proof obligation is that the interpretation of m in OCL agrees with its translation in PVS *and* that the translation is type consistent if and only if the interpretation of m in OCL is not undefined. As an example, we show the case of integer division /.

The semantics of the operator / and its translation $trans(/) = /$ agree on real numbers. We have to prove that in $t_0./(t_1)$ it holds: $\text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(t_1) \neq 0$. Since $trans(t_0./(t_1)) = trans(t_0)/trans(t_1))$ is well typed in PVS, there exists a proof of $[\![trans(t_1)]\!]\eta \neq 0$. From the induction hypothesis $\text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(t_1) = [\![trans(t_1)]\!]\eta$ we infer $\text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(t_1) \neq 0$. Ergo, $\text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(t_0./(t_1)) = [\![trans(t_0/t_1)]\!]\eta$ holds.

10. The proof for the case of *collection property call expressions* is analogous to the case of *operation property call expressions* (Item 9).

11. If there are OCL expressions t_0, t, and t_n, types T and T_0, and variables v_0 and a such that $\varphi = t_0 \to \mathit{iterate}(v_0 : T_0; a : T = t \mid t_n)$ and

$$\begin{aligned}\mathit{trans}(t_0 \to \mathit{iterate}(v_0 : T_0; a : T = t \mid t_n)) = \\ \text{iterate}(\mathit{trans}(t_0), \mathit{trans}(t), \text{LAMBDA } (v_0 : \mathit{trans}(T_0), a : \mathit{trans}(T)) : \\ \mathit{trans}(t_n)) \quad ,\end{aligned}$$

then, using the assumption that the translation is well-typed, it holds, that

a) $\mathit{trans}(t_0)$ is well typed, and by definition of iterate, see Equation (4.1), a *finite* collection.

b) $\mathit{trans}(t)$ is well typed, therefore its value is in $\mathfrak{D}(\mathit{trans}(T)) = \mathfrak{D}(T)$.

c) LAMBDA $(v_0 : \mathit{trans}(T_0), a : \mathit{trans}(T)) : \mathit{trans}(t_n))$ is well typed, therefore its value is in $\mathfrak{D}(\mathit{trans}(T') \to \mathit{trans}(T) \to \mathit{trans}(T))$. As a consequence of the induction hypothesis $\lambda x.\lambda y.\, \text{eval}(D, \overleftarrow{\sigma}, \sigma, E\{v_0 \mapsto x, a \mapsto y)(\varphi') \in \mathfrak{D}(T' \to T \to T)$.

Consequently, $\mathit{trans}(\varphi) \in \mathfrak{D}(T)$.

12. Let t be an OCL expression such that $\text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(t) = [\![\mathit{trans}(t)]\!]\eta$ holds as induction hypothesis. Using an induction over the definition of *flatten*, see Item 12 of Definition 2.19, and flatten, see Definition 4.2, we conclude

$$\text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(t \to \mathit{flatten}()) = [\![\mathit{trans}(t \to \mathit{flatten}())]\!]\eta \quad .$$

13. For all OCL type expressions T the propositions

$$\text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(T.\mathit{allInstances}()) = [\![\mathit{trans}(T.\mathit{allInstances}())]\!]\eta$$

and

$$\text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(T.\mathit{allInstances}@\mathbf{pre}()) = [\![\mathit{trans}(T.\mathit{allInstances}@\mathbf{pre}())]\!]\eta$$

are a direct consequence of the Definition 2.19 and Lemma 4.5.

In any case, the claim holds. □

We now have the tools to prove the next lemma, which strengthens the property formulated in Lemma 4.6. The preceding lemma states, that if the translation of a constraint is well-typed in PVS, then the constraint has the same value as the translation in PVS. Next, we show that, if a constraint is well-typed in OCL and its value is not undefined, then it has the same value as its translation to PVS.

Corollary 4.7. *Let D be a class diagram and σ and $\overleftarrow{\sigma}$ be two object diagrams forming an action. Then all OCL expressions t such that* $\mathrm{eval}(D, \overleftarrow{\sigma}, \sigma, \epsilon)(t) \neq \mathcal{J}$ *holds satisfy* $\mathrm{eval}(D, \overleftarrow{\sigma}, \sigma, \epsilon)(t) = [\![\mathit{trans}(t)]\!]$.

Remark 4.1. It is important to observe that Corollary 4.7 does not state anything about the provability of the type correctness constraints in PVS. There is indeed no guarantee that all true type correctness constraint have a proof, because the logic of PVS is incomplete in standard models.

Next, we are going to establish the first soundness result:

Theorem 4.8. *Let t be an OCL constraint such that for all object diagrams $\overleftarrow{\sigma}, \sigma$ forming an action such that* $\mathrm{eval}(D, \overleftarrow{\sigma}, \sigma, \epsilon)(t) \neq \mathcal{J}$ *holds. Then* $\vdash \mathit{trans}(t)$ *implies* $\mathrm{eval}(D, \overleftarrow{\sigma}, \sigma, \epsilon)(t) = \mathrm{true}$.

Proof. Using Assumption 4.3 we conclude from $\vdash \mathit{trans}(t)$, that $\mathit{trans}(t)$ evaluates to true. Using Lemma 4.7 we conclude $\mathrm{eval}(D, \overleftarrow{\sigma}, \sigma, \epsilon)(t) = \mathrm{true}$. □

Observe that the assumption $\mathrm{eval}(D, \overleftarrow{\sigma}, \sigma, \epsilon)(\varphi) \neq \mathcal{J}$ is crucial in Theorem 4.8. This assumption mainly excludes constraints like the one of Example 4.3, whose value is undefined in OCL, but true in PVS.

Example 4.3. Consider the constraint $\mathit{Integer.allInstances}() \to \mathit{forAll}(\mathrm{true})$. We have that the following holds: $\mathit{trans}(\mathit{Integer.allInstances}() \to \mathit{forAll}(\mathrm{true})) = \forall(x : \mathrm{int}) : \textsc{true}$. This constraint is provable in PVS:

$$\frac{\vdash \forall(x : \mathrm{int}) : \textsc{true} \quad x' : \mathrm{int}}{\textsc{true}}\ \text{Skolem}$$

On the other hand, we have $\mathrm{eval}(D, \overleftarrow{\sigma}, \sigma, \epsilon)(\mathit{Integer.allInstances}() \to \mathit{forAll}(\mathrm{true}) = \mathcal{J}$ for all class diagrams D and all object diagrams $\overleftarrow{\sigma}, \sigma$, because the set of integers, described as $\mathit{Integer.allInstances}()$ in OCL, is an infinite set. ♦

The previous example demonstrates that the definition of quantifiers in OCL is finitary, while PVS does not have this limitation. In practise, the interpretation of quantifiers in PVS is preferable, because it allows to quantify over, for example, all integers.

Theorem 4.9. *Let φ be an OCL constraint such that* $\vdash \mathit{trans}(\varphi)$. *Then for all object diagrams $\overleftarrow{\sigma}, \sigma$ we have* $\mathrm{eval}(D, \overleftarrow{\sigma}, \sigma, \epsilon)(\varphi) \in \{\mathrm{true}, \mathcal{J}\}$.

Proof. Assume $\mathrm{eval}(D, \overleftarrow{\sigma}, \sigma, \epsilon)(\varphi) = \mathrm{false}$. Then $[\![\mathit{trans}(\varphi)]\!] = \mathrm{false}$ by Corollary 4.7. Assuming the soundness of PVS and the fact $\vdash \mathit{trans}(\varphi)$, we also have $[\![\mathit{trans}(\varphi)]\!] = \mathrm{true}$, which is a contradiction. Therefore $\mathrm{eval}(D, \overleftarrow{\sigma}, \sigma, \epsilon)(\varphi) \neq \mathrm{false}$. □

By Example 4.3 we already know of constraints which result in undefined in OCL. However, only the choice of translating *forAll* and *exists* to their respective quantifiers cause this problem.

4.7 Summary and Conclusion

The formalism and the methods described in this chapter have been implemented in a prototype compiler. We have tested it by, for example, verifying the object-oriented version of the *Sieve of Eratosthenes* described in Section 4.4. A trace-based proof is described in Chapter 6. Another proof using the compiler, which also establishes the liveness property that every prime will eventually be generated, has been established by Tamarah Arons using TL-PVS [129].

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 concept of messages preceding one another, crucial for the sieve, is 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.[7] If one of the two properties does not hold, then the sieve does not satisfy its specification.

The run-time of the translator is usually less than a minute. The runtime is dominated by the time required to prove the model correct in PVS in any case. The coarse level of specifications in OCL and the expressive power of OCL is not sufficient to automate the whole verification process. For the proofs, annotations of the states of a state machine expressing invariants are highly useful, because 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 [45] for references) as implemented in [43].

The work reported by Traore [148] defines a formalisation of UML state machines in PVS, which has been implemented in the PrUDE program. This program also implements a translation of state machines into PVS, but the semantics of state machines used by Traore differs from ours. This program also lacks the translation of OCL constraints to PVS.

The USE tool, described in, among others, the thesis of Mark Richters [133], implements an interpreter of OCL for run-time checking. USE only implements a way for testing OCL constraints, but not for verifying them using a theorem prover.

Brucker and Wolff describe a formalisation of OCL in Isabelle/HOL [101], another theorem prover for higher order logic [19, 20]. Contrary to our approach, they have extended partial functions to extended total functions by introducing an undefined value. This entailed the extension of the logic used in Isabelle/HOL to a three-valued logic, amounting to a deep embedding of OCL in the theorem prover. While this approach allows the verification of meta theorems, the verification of an actual UML model with respect to its OCL specification still requires the additional proof that all values do not correspond to undefined.

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