Reasoning about Java Classes
(Preliminary Report) *

Bart Jacobs, Joachim van den Berg,
Marieke Huisman, Martijn van Berkum
Dep. Comp. Sci., Univ. Nijmegen,
P.O. Box 9010, 6500 GL Nijmegen, The Netherlands.
{bart,joachim,marieke,mvberkum}@cs.kun.nl

Ulrich Hensel, Hendrik Tews
Inst. Theor. Informatik, TU Dresden, D-01062 Dresden, Germany.
{hensel,tews}@tcs.inf.tu-dresden.de

Abstract

We present the first results of a project called LOOP, on formal methods for the object-oriented language Java. It aims at verification of program properties, with support of modern tools. We use our own front-end tool (which is still partly under construction) for translating Java classes into higher order logic, and a back-end theorem prover (namely PVS, developed at SRI) for reasoning. In several examples we demonstrate how non-trivial properties of Java programs and classes can be proven following this two-step approach.

1 Introduction

Being able to reason about programs has always been one of the central objectives of research in computer science. Progress in this area is slow, because the subject matter is complicated. In order to reason about a program, one first has to assign meaning to this program (usually as some function acting on states), and then reason (using a suitable logic) about what this program does. Such reasoning is often subtly different from ordinary mathematical reasoning because of typical imperative phenomena, like side-effects, or because of different forms of partiality (ordinary or abrupt termination, *e.g.* via exceptions).

This paper concentrates on reasoning about Java [5, 10]. Java is quickly becoming one of the most widely used programming languages. Being able to reason about programs and classes in Java—and hence being able to establish correctness or incorrectness of a Java implementation with respect to some specification, see explicitly in Subsection 4.6—is of considerable interest. We use a proof tool (namely PVS [22]) for reasoning, and avoid arguments "by hand"—which are generally considered less trustworthy. Using such a proof assistant in this area has definite advantages.

- A proof tool keeps track of which results have and which have not been proved. It can easily tell a user if all the assumptions on which a certain result relies have been proved. These are typical bureaucratic activities, which can best be done by tools, because they often lead to mistakes, when done by humans. Other such bookkeeping activities include keeping track of all case distinctions in a proof—which there are usually many, when reasoning about programs.
- Program verification involves much routine equational and Boolean reasoning. A tool can do this very well, once it has been loaded with appropriate rewrite rules (and decision procedures).
- Side-effects are important when dealing with imperative programs. However, they are notoriously hard to reason about. Once they are properly incorporated in PVS theories, the proof tool helps the user to keep track of all these side-effects, and to make the right deduction steps.
- Side-conditions which are required to hold before an auxiliary lemma can be applied are enforced by the tool. This helps to prevent small mistakes.

In brief, a proof assistant is like a "sceptic colleague" who patiently checks all details and is willing to do routine tasks.

An assertional approach (as used by such a proof tool, or possibly also by someone reasoning by hand) has a definite advantage over testing: by testing one only checks a limited number of cases[1]. In contrast, using assertional methods one can prove statements of the form: *for all* parameters it is the case that $\cdots$. This achieves an appropriate level of generality (and thus, confidence).

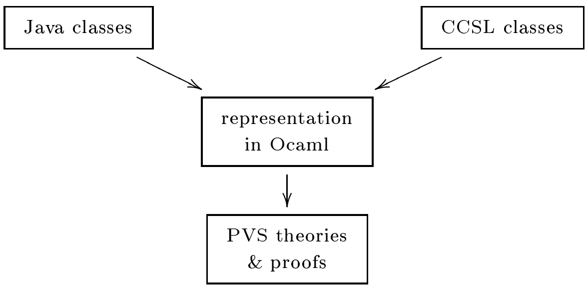
So far we have discussed the use of a proof assistant in our project. Such a tool is used as a back-end, to our own tool,

*In: *Object-Oriented Programming Systems, Languages and Applications*, ACM Press, p.329–340, 1998.

[1] Selecting appropriate test cases is indeed a major issue in this area.

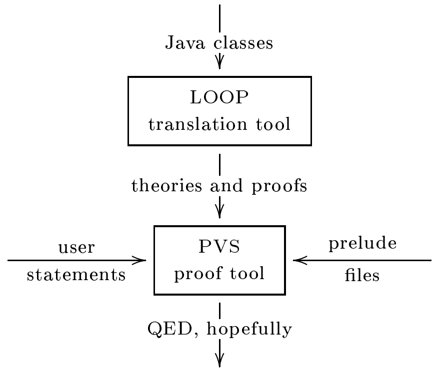
which we call LOOP (for Logic of Object-Oriented Programming). The LOOP tool translates Java classes into higher order logic, thus providing input for the back-end proof tool PVS. In translating Java classes to logic, the LOOP tool provides a logical semantics for Java. This will be an important topic in the paper. The LOOP tool is still under development, but what we discuss here is a version which automatically translates a non-trivial part of Java. For example, it handles inheritance and late binding in Java classes, but it does not handle threads.

What we shall describe is a part of a more general LOOP tool described in [13] for translating object-oriented specifications into higher order logic. The tool performs the following transformations.



First it reads (lexes and parses) classes in CCSL (Coalgebraic Class Specification Language, see [13]) or Java, and transforms these in some internal representation in the programming language Ocaml [24, 20] (the implementation language of LOOP). This representation is subjected to certain internal analyses, *e.g.* for establishing the inheritance relationships between classes. Finally, it is transformed into theories (and proofs) of the PVS theorem prover [22]. Much of the internal code of the tool is shared for both the translations from CCSL and Java classes to PVS. The earlier paper [13] on CCSL showed the merits of a coalgebraic approach in the mathematically clean world of specification. Here, in contrast, we show that the coalgebraic framework can properly handle the messy details that come with an existing real-world object-oriented programming language (for which we use Java as an example). In doing so, we use many of the semantical ideas and results that have been developed over the past few decades.

This LOOP tool (on Java classes) is typically used as follows. Assume a user wishes to prove a certain property about a particular Java class (or about a collection of classes). For example, that a certain method always terminates normally, or that some property is an invariant of a class. The user can run the LOOP tool on the class[2], say with name `MyClass`. The tool then produces a new file, called `MyClass_basic.pvs`[3]. It contains a translation into the higher order logic of PVS of the Java class in the original file. This forms the basis for the user's own work: (s)he can now create a separate file, say `MyClass_user.pvs`, with user-defined theories in which the generated theories from `MyClass_basic.pvs` are imported. Here the statements that the user wishes to prove about the (translated) Java class should appear. All these (`.pvs`) files can be loaded into the PVS theorem prover, and the user can start trying to prove the desired results, using result of the translation of the LOOP tool. Summarising:



In this paper we describe several examples of this two-step approach, usually by presenting: (1) the original Java class(es) on which the LOOP tool is run, (2) some relevant details of the resulting translation, (3) some propositions in PVS that we wish to prove, and possibly, (4) some details of the actual proof. See the above diagram in which these four points can be recognised.

This project makes heavy use of traditional results and techniques from the semantics of programming languages, see *e.g.* [6, 21, 12, 26]. In a nutshell, traditional reasoning about programs in a language $\mathcal{L}$ proceeds as follows. First, a suitably rich mathematical structure $\mathcal{D}$ is identified, which can serve as semantic domain for $\mathcal{L}$, and as domain of reasoning. Then, an interpretation function $[\![-]\!]\colon \mathcal{L} \to \mathcal{D}$ is written out, mapping the well-formed expressions of $\mathcal{L}$ to elements of the domain $\mathcal{D}$. Usually, this interpretation function is "compositional", so that the interpretation $[\![s_1; s_2]\!]$ of a composite statement $s_1; s_2$ is equal to $[\![s_1]\!] \# [\![s_2]\!]$, where $\#$ is a composition operation defined on $\mathcal{D}$. Once this interpretation function is given, one can prove properties about statements s in $\mathcal{L}$ by reasoning about $[\![s]\!]$ in $\mathcal{D}$.

Basically the same approach forms the basis of the translation of Java classes into the higher order logic of PVS, as performed by the LOOP tool. But also, there are some notable differences:

- The semantic domain $\mathcal{D}$ is not described in the ordinary language of mathematics, but in the logic and type theory of PVS. These descriptions form part of certain "prelude" PVS files, which are common to all translations, see Section 2.

- The interpretation function $[\![-]\!]$ is not written out by hand, but calculated by the LOOP tool. How to

[2]We always assume that Java classes which are fed into the LOOP tool are accepted by a (standard) Java compiler. Also, at this stage, all Java classes that have to be translated are required to be in one input file for the LOOP tool.

[3]Actually, it also produces a file `MyClass_basic.prf`, containing proofs of standard results in the file `MyClass_basic.pvs`. This proof file is not relevant here.

do this translation is of course a major issue in this project. Section 3 gives more information.

- Proofs about the resulting interpretation are not done by hand but by using a theorem prover (in this case PVS), as already discussed above.

- The translation works for object-oriented programs, organised in classes. Therefore, the relations between classes (inheritance "is a subclass of", and aggregation "is a component of") have to be translated appropriately, so that operations from one class are available (if needed) in another. This is based on a coalgebraic analysis of classes, see *e.g.* [23, 17, 18].

- Certain additional definitions are generated automatically. Especially, for each class appropriate notions of invariance and bisimilarity are generated, see Subsection 4.5. These notions make it easier for the user to express certain results.

- For each field and method in a translated Java class appropriate rewrite lemmas are generated. These greatly improve automatic reasoning.

What distinguishes the current project from other formal approaches to object-orientation (see *e.g.* [1, 2]) is the combination of a coalgebraic semantics of classes, and extensive use of tools, both for translating and for reasoning. The use of coalgebras emphasises that a state space forms a black box with only limited access to its elements. This naturally leads to notions of bisimilarity (observational indistinguishability) and invariance—for reasoning about objects—which are fundamental in the theory of coalgebras, see *e.g.* [19, 18, 25]. Also, modeling statements and expressions as coalgebraic state transformer functions (see in the next section below) allows a natural treatment of non-termination (which is non-trivial in algebraic setting) and also of abrupt termination (*e.g.* caused by exceptions).

The logic that we use for reasoning about Java classes is (ultimately) the higher order, classical logic of the proof tool PVS. Current work involves incorporation of Hoare logic [15, 3], via appropriate definitions and rules in PVS. This will enable us to use standard reasoning techniques in program verification, including, for example, variants and invariants for while loops ([16]). In the current paper we shall not use Hoare logic, and our reasoning is directly based on definitions.

The work on the LOOP project can be divided into the following categories (with initials of the authors who contribute most to these parts): (1) Java semantics (BJ, MH), (2) automatic translation (JvdB, MvB, BJ), (3) proofs and proof methods (MH, BJ), (4) general LOOP infrastructure (UH, HT). We should emphasise that this is very much work in progress, and that we are nowhere near a complete translation of all possible Java classes. The most important restrictions are discussed in Section 3. But, as we hope that the examples below demonstrate, we can already handle a substantial part of the language. The manner in which it is done is of general interest, and the main topic of this paper.

So far we have tested our tool only on microscopic examples: our tests are not as big as possible, but as sick as possible. A semantics for a language like Java should of course also include programs which are generally considered bad style (*e.g.* because of multiply occurring variable names, or control flow via too many `return`'s, `break`'s, `continue`'s or exceptions). It turns out that such programs are difficult to reason about—and therefore good test programs for our approach. Formal verification of software will always remain a knowledge and labour intensive activity, but we hope that (eventually) our tool can contribute to this area. For example, it may become worthwhile to formally verify certain properties of classes which go into standard class libraries or into safety critical applications.

This paper is organised as follows. It starts with a brief description of the logical semantics that is used for Java. Subsequently, in Section 3 the translation that is performed by the LOOP tool is sketched. Both these are substantial topics, which we can only touch upon in the current paper. The remainder of the paper is devoted to (typical) examples. It discusses reasoning about a non-trivial method body, inheritance and late binding, local variables and recursion, while loops with breaks and continues, invariance results, and also, briefly, component classes.

2 Java semantics in the higher order logic of PVS

In this section we give an impression of the "prelude" PVS files which provide the background for the translation of Java classes into higher order logic. They incorporate the semantic structure $\mathcal{D}$, as discussed in the previous section. This prelude is divided into five files, describing the relevant datatypes, statements, expressions, operations, and the underlying memory model. The total size of these PVS files is just under 200K (about 6000 lines). We concentrate on some essential ingredients.

Two important syntactic categories in Java are statements and expressions. These will both be translated as state transformer functions (in PVS), namely as:

```
[Self -> StatResult?[Self]]
            and
[Self -> ExprResult?[Self, Out]]
```

The type `Self` is a parameter for the underlying state space[4], and `Out` is the parameter type of the output of the expression. We frequently use the question mark `?` in PVS expressions on which our translation is based because `?`'s cannot occur in Java keywords nor in Java identifiers, and so we can prevent name clashes. It clutters up the notation a bit, but it probably is best simply to ignore all these `?`'s. A more important point is that these state transformer functions are examples of coalgebras (see [19]): they have a structured type (`StatResult?` or `ExprResult?`, and not `Self`) as codomain. Such functions cannot be written down in algebraic approaches (where one typically has structured types

[4] In the actual translation, `Self` will be instantiated as `OM?`, describing the global memory, see the end of this section.

```
PreStatResult?[Self, Abnormal : TYPE] : DATATYPE
BEGIN
  hang? : hang??
  norm?(ns? : Self) : norm??
  abnorm?(dev? : Abnormal) : abnorm??
END PreStatResult?

PreExprResult?[Self, Abnormal, Out : TYPE] : DATATYPE
BEGIN
  hang? : hang??
  norm?(ns? : Self, res? : Out) : norm??
  abnorm?(dev? : Abnormal) : abnorm??
END PreExprResult?

ExprAbn?[Self : TYPE] : DATATYPE
BEGIN
  IMPORTING ExceptionInterpretation
  excp?(es? : Self, ex? : ExcpIds) : excp??
END ExprAbn?

StatAbn?[Self : TYPE] : DATATYPE
BEGIN
  IMPORTING ExceptionInterpretation, Lift?[string]
  excp?(es? : Self, ex? : ExcpIds) : excp??
  rtrn?(rs? : Self) : rtrn??
  break?(bs? : Self, blab? : Lift?[string]) : break??
  cont?(cs? : Self, clab? : Lift?[string]) : cont??
END StatAbn?
```

Figure 1: The four main datatypes used for the translation

as domains). The approach we use to give semantics to Java (as implemented in the LOOP tool) is based on coalgebras, and is thus perfectly able to handle such (statement and expression) functions.

The above codomain types of statements and expressions are abbreviations, obtained by substitution:

```
StatResult?[Self]
  =  PreStatResult?[Self, StatAbn?[Self]]
ExprResult?[Self, Out]
  =  PreExprResult?[Self, ExprAbn?[Self], Out]
```

involving four data types in PVS, see Figure 1. The two output types `PreStatResult?` and `PreExprResult?` describe the possible outcomes of statements and expressions, as state transformer functions. A (translated) statement in a particular state can either hang (yield outcome `hang?`), terminate normally (with outcome `norm?(x)`, where `x` is a new state in `Self`), or terminate abnormally/abruptly (with outcome `abnorm?(y)`, with `y` describing the kind of abnormality). The latter is used to model exceptions and statements affecting the control flow like break, return, and continue. In contrast an outcome `hang?` corresponds to non-termination.

The PVS expressions `hang?`, `norm?` and `abnorm?` are the constructors of the datatype `PreStatResult?`. The associated recognisers are `hang??`, `norm??` and `abnorm??`, telling whether an element in `PreStatResult?` is of the form `hang?`, `norm?(x)` or `abnorm?(y)`, respectively. The associated accessors are `ns?` (extracting the `x` in `norm?(x)`) and `dev?` (extracting `y` in `abnorm?(y)`). The outcome of a (translated) expression is very similar, except that normal termination produces a (new) state—because expressions can have side-effects—together with a result in the output type `Out` of the expression. Hence there is a *binary* constructor `norm?` in the data type `PreExprResult?`, with a state in `Self` and a result in `Out` as arguments. In these definitions it is convenient to keep a type `Abnormal` of abnormalities as parameter. The standard instantiation for `Abnormal` in `PreStatResult?` is the type `StatAbn?` describing the possible abnormalities for statements. These include exceptions, returns, breaks and continues. The latter two statements may occur in Java with or without a label. This possibility is captured in our translation by using a `Lift?[string]` argument for the break and continue constructors `break?` and `cont?` in `StatAbn?`—where the type `Lift?[X]` adds a (new) bottom element to an arbitrary type `X`. In a similar fashion, the type `ExprAbn?` captures abnormalities in expressions (namely, exceptions only).

On the basis of these datatypes we can already introduce some basic program constructs. For example, a composition (infix) operator `#` (intended as translation of `;` in Java) is defined in PVS. It takes two statements `s`, `t` of type `[Self -> StatResult?[Self]]` and produces a new statement `s # t` describing `s` followed by `t`, again of type `[Self -> StatResult?[Self]]`. It is defined as:

```
(s # t) : [Self -> StatResult?[Self]] =
  LAMBDA(x : Self) :
    IF norm??(s(x))
    THEN t(ns?(s(x)))
    ELSE s(x)
    ENDIF
```

Thus if `s` terminates normally in state `x`, resulting in a next state `y = ns?(s(x))`, then `(s # t)(x)` is `t(y)`. And if `s` hangs or terminates abnormally in state `x`, then `(s # t)(x)` is `s(x)` and `t` is not executed. It is not hard to show that `#` is associative, and has a (left and right) unit `skip?`, given by `skip?(x) = norm?(x)`. Hence statements form a monoid.

In a similar manner we define a conditional statement, written as `IF?THEN_ELSE(c)(s)(t)`, for a Boolean expression `c` of type `[Self -> ExprResult?[Self, bool]]` and two statements `s`, `t`, as:

```
IF?THEN_ELSE(c)(s)(t) : [Self -> StatResult?[Self]] =
  LAMBDA(x : Self) :
    IF hang??(c(x))
    THEN hang??
    ELSIF norm??(c(x))
    THEN IF res?(c(x))
         THEN s(ns?(c(x)))
         ELSE t(ns?(c(x)))
         ENDIF
    ELSE abnorm?(excp?(es?(dev?(c(x))),
                       ex?(dev?(c(x)))))
    ENDIF
```

In this manner, Java constructs are translated in the prelude files, following the explanations in [10]. Another such example is a `RETURN?` statement in PVS, defined as:

```
RETURN? : [Self -> StatResult?[Self]] =
  LAMBDA(x : Self) : abnorm?(rtrn?(x))
```

It creates a return abnormality, see Figure 1. There are analogous statements for throwing other abnormalities. Similarly, the two conjunction operators `&` and `&&` of Java are translated as `AND` and `ANDTHEN` in PVS, respectively. They are defined on Boolean expressions `e`, `d` as:

```
(e AND d) : [Self -> ExprResult?[Self, bool]] =
  LAMBDA(x : Self) :
    IF norm??(e(x))
    THEN LET y = ns?(e(x)), r = res?(e(x)) IN
         IF norm??(d(y))
         THEN norm?(ns?(d(y)), r AND res?(d(y)))
         ELSE d(y)
         ENDIF
    ELSE e(x)
    ENDIF

(e ANDTHEN d) : [Self -> ExprResult?[Self, bool]] =
  LAMBDA(x : Self) :
    IF norm??(e(x))
    THEN LET y = ns?(e(x)), r = res?(e(x)) IN
         IF NOT r
         THEN e(x)
         ELSE d(y)
         ENDIF
    ELSE e(x)
    ENDIF
```

Notice how side-effects are propagated through these composite expressions, via the intermediate state `y`. Both `AND` and `ANDTHEN` equip the type of Boolean expressions with a monoid structure (both with the constantly true expression as unit).

Similar, but more complicated translations are formulated for switch, while, *etc.* The latter works on a Boolean expression and a statement, and basically iterates the statement a certain number of times, in case there is an n such that after n iterations the expression becomes false, or an abrupt termination occurs. If there is no such n, the while statement hangs. We can declaratively make this distinction in logic. More details are given in Subsection 4.4.

We describe a few more functions, that will be used below. Every element `a:Out` of an output type `Out` is turned into a "constant" expression of type `Out` via the definition:

```
const?(a) : [Self -> ExprResult?[Self, Out]] =
  LAMBDA(x : Self) : norm?(x, a)
```

We have a special function `E2S?` for transforming an expression `e` into a statement, essentially by forgetting the output, if any.

```
E2S?(e) : [Self -> StatResult?[Self]] =
  LAMBDA(x : Self) :
    IF hang??(e(x))
    THEN hang?
    ELSIF norm??(e(x))
    THEN norm?(ns?(e(x)))
    ELSE abnorm?(excp?(es?(dev?(e(x))),
                       ex?(dev(e(x)))))
    ENDIF
```

Certain constructs in Java may be used both as expressions and as statement. These constructs are then standardly translated as expressions, and `E2S?` is applied to them when they are used as statements. There is another useful (infix) operation `##`, which produces an expression `e ## d` from two expressions `e` and `d` by applying `d` to the (normal) state produced by `e`, while keeping the result of `e`.

```
(e ## d) : [Self -> ExprResult?[Out]] =
  LAMBDA(x : Self) :
    IF norm??(e(x))
    THEN LET y = ns?(e(x)), r = res?(e(x)) IN
         IF norm??(d(y))
         THEN norm?(ns?(d(y)), r)
         ELSE d(y)
         ENDIF
    ELSE e(x)
    ENDIF
```

This operation `##` is used in the translation of "post" operations (such as post-increment `i++`), where first a value is produced and then some operation is performed. Examples will appear below.

A large part of our prelude files is devoted to suitable rewrite lemmas for all these definitions. They enable PVS to handle substantial parts of proofs automatically via rewriting. These lemmas are usually very easy and follow almost directly from the definitions, but they are extremely useful for automatic reasoning.

In one of the prelude files a model `OM?` of an object memory is defined, containing an infinite[5] number of memory cells, each capable of storing an object. It comes equipped with operations for reading and writing values and references at particular positions. All classes are translated as coalgebras, see the next section, acting on (a position in) this global state space `OM?`.

3 Translating Java classes

The LOOP tool calculates a function $[\![-]\!]$ which assigns meaning to Java classes. It follows the Java grammar [10, Chapter 19], and takes *e.g.* $[\![$`e1 & e2`$]\!] = [\![$`e1`$]\!]$ `AND` $[\![$`e2`$]\!]$ as PVS translation of `e1 & e2` in Java. Such clauses are handled, one-by-one, by Ocaml's yacc. Basically, this is how the translation works. But there is much more to say.

Ignoring static initialisers, a class in Java consists of fields, methods and constructors. The latter are not translated yet, but seem to present no fundamental difficulties, so we concentrate on fields and methods. The fields (sometimes called instance variables) and methods of a class are collected by the LOOP tool in a PVS interface type (like in [13]). For each field `i` an associated assignment operation `i_becomes` is generated. Thus, a class

```
class MyClass {
  byte i, j;
  void stat_meth() { .. }
  float expr_meth() { .. }
}
```

will give rise to the following interface (record) type in PVS.

```
MyClassIFace[Self] =
  [#
    i : byte,
```

[5]Hence we do not bother about garbage collection, and an `OutOfMemoryException` is never thrown in our translated classes.

```
    j : byte,
    i_becomes : [byte -> Self],
    j_becomes : [byte -> Self],
    stat_meth : StatResult[Self],
    expr_meth : ExprResult[Self, float]
  #]
```

A coalgebra for class MyClass is a function in PVS of the form

```
c : [Self -> MyClassIFace[Self]]
```

where MyClassIFace[Self] is the interface type generated for class MyClass. Such a coalgebra thus contains all the operations of a class in a single function. The individual operations can be extracted via (automatically generated) definitions like:

```
i(c) : [Self -> byte] =
  LAMBDA(x : Self) : i(c(x))
stat_meth(c) : [Self -> StatResult[Self]] =
  LAMBDA(x : Self) : stat_meth(c(x))
```

In the sequel we shall always use individual operations with respect to such a coalgebra c. For more information about coalgebras (versus algebras), see [19].

Notice that the above field i(c) in a coalgebra c has type [Self -> byte]. We would like to use it as an expression, with type [Self -> ExprResult?[Self, byte]]. This is achieved by translating every occurrence of i (in the body of a Java method) as F2E?(i(c)), where F2E? is a function—whose name is an abbreviation of field-to-expression—that is defined for an arbitrary result type Out as:

```
F2E? : [[Self -> Out] ->
        [Self -> ExprResult?[Self, Out]]] =
  LAMBDA(f : [Self -> Out]) :
    LAMBDA(x : Self) :
      norm?(x, f(x))
```

Similarly, the associated assignment operation i_becomes(c) has type [Self, byte -> Self]. We would like to use it as acting on an expression of type byte and producing an expression[6] with result type byte. For this we use a function A2E?, standing for assignment-to-expression. It is defined as:

```
A2E? : [[Self, Out -> Self] ->
        [[Self -> ExprResult?[Self, Out]] ->
         [Self -> ExprResult?[Self, Out]]]] =
  LAMBDA(a : [Self, Out -> Self]) :
    LAMBDA(e : [Self -> ExprResult?[Self, Out]]) :
      LAMBDA(x : Self) :
        IF norm??(e(x))
        THEN LET y = ns?(e(x)), r = res?(e(x)) IN
             norm?(a(y, r), r)
        ELSE e(x)
        ENDIF
```

The body of a method meth in class MyClass gives rise to a predicate on a MyClass-coalgebra c which expresses that meth(c) is equal to the (translation of the) body of meth. That is, a method

```
void move(int da, int db) {
  fst = fst + da;
  snd = snd + db;
}
```

in a class with integer fields fst and snd, is translated into a predicate called move_def? on c, which expresses that for all states x,

```
FORALL(da : int_java, db : int_java) :
  move(c)(x, da, db) = (
    E2S?(A2E?(fst_becomes(c))
             (F2E?(fst(c)) + const?(da))) #
    E2S?(A2E?(snd_becomes(c))
             (F2E?(snd(c)) + const?(db)))
  )(x)
```

The use of all these auxiliary functions E2S?, A2E?, F2E? looks a bit artificial in this simple example, because everything terminates normally and there are no side-effects. But that is not typical. (Actually, this is a simplified version of the real translation which additionally involves a LET construct for handling the parameters da, db, see Figure 2 for a complete translation. The above serves to convey the main idea.)

All method definitions are thus translated into predicates. The latter are combined via conjunction into a single predicate MyClassAssert? on the coalgebra c. A user can then develop the theory of coalgebras satisfying such predicates, incorporating how methods are implementated. These coalgebras can be seen as models of the class. This is basically as in [13].

This account simplifies matters slightly for explanatory purposes. We have already mentioned (at the end of the previous section) that all coalgebras operate on the same state space OM?, describing a global memory. Each field declaration, say int i, is implemented via a function, called get?int which can read a value at a particular location in OM?. The associated assignment operation (i_becomes) writes at this same location, via a function put?int. This memory location is a PVS variable p in the predicates defining variables, assignments and method implementations, and ultimately also MyClassAssert?. Hence as models of classes we really use functions in a dependent product of the form:

```
d : [p : nat -> (MyClassAssert?(p))]
```

so that d(p) is a coalgebra satisfying the method implementations. (It should have been used instead of c above.) One can understand d(p) as an implementation of the class MyClass which acts on memory location p. In general, if a new variable of a class is created, it gets a (new) position p in the main memory OM? together with a coalgebra (of the class) acting on this position p. An object is then basically a reference to a memory cell.

Here we conclude our brief sketch of the translation that the LOOP tool performs. We emphasise that this translation is far from complete. For example, it does not handle threads, and some of the language constructs are not covered yet (like constructors). However, the tool translates many statements and expressions already. Besides being incomplete, the translation also simplifies matters. For example, both floating point types float and double in Java are translated to the PVS type real. The latter is introduced axiomatically in PVS, and is for example dense and equipped with a complete order $\leq$. The former are approximations of real numbers (described precisely in the IEEE 754 floating point format). In order to translate accurately,

[6] Assignments in Java can be expressions, like j = 5 in i = (j = 5).

one would have to formalise this IEEE format in PVS. This is a non-trivial exercise, which is a project on its own, see *e.g.* [7]. Similarly, we translate all Java integer types (`byte`, `short`, `int`, `long` and `char`) to the PVS type `int` of integers, without taking bounds into account. Another (temporary) simplification involves exceptions. These are translated as sets of natural numbers (*e.g.* `IndexOutOfBoundsException` is {`x:nat|50 <= x AND x < 60`}, which is a completely arbitrary choice). Catching an exception then involves checking a subset relationship. This simplification works well in many situations because an exception object (like the `e` in the `WeirdExpr` class in the next section) is rarely really used (with as possible exception, in a print statement). The motivation behind these simplifications is to be able to get a rudimentory translation off the ground, and not to be held up by initially irrelevant details.

4 Examples

In this section we elaborate some examples. In particular, we discuss: several Java classes, their translation into PVS by the LOOP tool, some results that a user may wish to prove (on the basis of the translation), and proofs of such results. Only the first example will be described in some detail.

Many of the proofs of the results about Java classes that we present below are done by automatic rewriting. This is very convenient and useful. But it is important to stress that this only works for "simple" (non-looping) statements, such as assignments, conditionals, or operations. The more difficult statements with while or for loops or with recursion still require steering of the user at essential points (*e.g.* in order to find loop invariants). But also within such proofs the routine calculations can be done by automatic rewriting.

4.1 A weird method

Consider the following (silly) Java class.

```
class WeirdExpr {
  int i;
  int lets_calculate(int j) {
    try { i *= (i > 5) ? (i++ % --j) : 8; }
    catch(Exception e) {
      j--;
      return i - j; }
  return i + j;
  }
}
```

It contains an integer field `i` and a method `lets_calculate` yielding an integer after some intricate computation involving a conditional operator `? :` and a remainder operation `%`. The latter throws an exception, if its second argument is 0. The computation in itself is uninteresting, but the challenge is to express the integer outcome (if any) of this method, in terms of the values of the parameter `j`, and the field `i`.

In order to determine this outcome we have to take the following into account (among many other things).

- The evaluation strategy: in the remainder expression `i++ % --j` one computes the value of `i` as first argument, then `i` is incremented, then `j` is decremented, and the resulting value (of `j`) is taken as second argument, so that the remainder can finally be determined.
- Exception handling: if the parameter `j` is `1`, then the Java remainder operation `%` (translated as `//`) throws an `ArithmeticException`, which is caught by the subsequent `catch` clause—because `ArithmeticException` is a subclass of `Exception`; this causes a particular flow of control. One of the subtleties in this example is that the increment expression `i++` only has a visible effect if the exception is thrown—because otherwise it is overruled by the `*=` assignment.
- Return handling: the first `return` statement causes a jump of control to the end of the method.

The latter two points are handled by the `abnorm?` option in statements and expressions (as discussed in the previous section). Special functions `TRY?CATCH` and `CATCH_EXPR_RETURN?` are defined which detect such abnormal outcomes. They remove certain abnormalities and take appropriate action. The first point is handled by suitable PVS representations of the (pre- and post-) increment/decrement and remainder operations, so that arguments are evaluated in the right order. See the analogous definitions of the `AND` and `ANDTHEN` functions in the previous section.

Running the LOOP tool on this class yields a series of PVS theories. They contain the translation of the method `lets_calculate`, given in Figure 2. This translation is probably unreadable, and not really meant for human consumption, but is included only to show what really comes out. Hopefully, the reader will recognise the main structure of this Java method in its PVS translation, *e.g.* the Java conditional operator `? :` translated as `QUESTION?` in PVS. It is not feasible to explain the whole translation in detail, so we will focus on some significant details.

- The `c` and `x` variables in `lets_calculate(c)(x, j)` on the left hand side of the equation refer to the coalgebra of the current class (*i.e.* of `WeirdExpr`) and the current state, respectively. Recall that methods and fields are always described with respect to some coalgebra.
- A special local variable `ret?lets_calculate` (together with an associated assignment) is used for the output result of this method (which is returned at the end, by the `CATCH_EXPR_RETURN?` function). Another special Java variable `par?j` holds the value of the PVS variable `j` (set at the beginning). It is used because PVS variables are different from Java variables (for which there are assignments)[7]. For both these variables a new memory cell is allocated at the next free position, given by `top?`—after which the `top?` is incremented by `topinc?`.

[7] Using such an auxiliary variable also ensures that parameters are passed by value, see [5, 2.6.1].

```
FORALL (j : int_java) :
  lets_calculate(c)(x, j) =
    LET ret?lets_calculate : [OM? -> int_java] = (LAMBDA (y: OM?) : get?int(y , top?(x) , 1)),
        ret?lets_calculate_becomes : [[OM? , int_java] -> OM?] = (LAMBDA (y: OM? , v: int_java) :
                                                          put?int(y , top?(x) , v , 1)),
        par?j : [OM? -> int_java] = (LAMBDA (y: OM?) : get?int(y , top?(x) , 0)),
        par?j_becomes : [[OM? , int_java] -> OM?] = (LAMBDA (y: OM? , v: int_java) :
                                                  put?int(y , top?(x) , v , 0))
    IN
      CATCH_EXPR_RETURN?
        (E2S?(A2E?(par?j_becomes)(const?(j))) #
         (TRY?CATCH
            ((E2S?(A2E?(i_becomes(c))((F2E?(i(c)) *
               QUESTION?((F2E?(i(c)) > const?(5)))
                        (((F2E?(i(c)) ## A2E?(i_becomes(c))(inc(F2E?(i(c))))) //
                          A2E?(par?j_becomes)(dec(F2E?(par?j)))))
                        (const?(8))))))
            ((: (Exception ,
                (E2S?((F2E?(par?j) ## A2E?(par?j_becomes)(dec(F2E?(par?j))))) #
                (E2S?(A2E?(ret?lets_calculate_becomes)((F2E?(i(c)) - F2E?(par?j)))) #
                 RETURN?))) :))) #
         (E2S?(A2E?(ret?lets_calculate_becomes)((F2E?(i(c)) + F2E?(par?j)))) #
         RETURN?))
     (ret?lets_calculate)(topinc?(put?empty(x? , top?(x?)) , 1))
```

Figure 2: The LOOP translation of the weird method in PVS

- A *pre* increment or decrement operation is translated simply by an assignment (which returns a value, see the previous section). For a *post* increment or decrement operation we use the ## operation between two expressions, as described in the previous section.
- The TRY?CATCH statement takes as argument a list of pairs—indicated in PVS by (: ··· :)—consisting of an exception class, together with the corresponding statement that should be executed if an exception of the kind in the first part of the pair occurs. In this example the list contains only one pair.

An example result that a user may wish to prove in PVS is the following.

```
p : VAR nat
d : VAR [m : nat -> (WeirdExprAssert?(m))]
x : VAR OM?

lets_calculate_return : LEMMA
  FORALL(j : int_java) :
    norm??(lets_calculate(d(p))(x, j))
      AND
    res?(lets_calculate(d(p))(x, j)) =
      IF i(d(p))(x) > 5
      THEN IF j = 1
           THEN i(d(p))(x) - j + 3
           ELSE i(d(p))(x) *
                remainder(i(d(p))(x), j-1) + j - 1
           ENDIF
      ELSE i(d(p))(x) * 8 + j
      ENDIF
```

The lemma states that for all integers j, running the method lets_calculate with respect to the WeirdExpr coalgebra d(p) (acting in memory location p) in state x with parameter j terminates normally (expressed by norm??(-)), and the resulting output value res?(-) satisfies the IF ··· THEN ··· ELSE clause. It expresses the outcome of the method run in state x in terms of the values of the field i in state x and of the parameter j. Notice that the result involves a universal quantifier FORALL. It achieves a level of generality which can never be obtained by simply testing (*i.e.* by running the method for specific values and checking the outcome). This shows the power of a theorem proving approach to formal verification.

The above lemma can be proved in PVS by using basically only two proof commands: (load-rewrite-theories ···) and (do-rewrite). All the expressions in the lemma are then suitably rewritten (following the evaluation strategy of Java, as incorporated in the definitions of the prelude, see Section 2) to the required result. This involves 222 single rewrite steps[8]. Such rewriting must be done in a clever manner, because the number of possibilities in each step is large: in principle, each expression and statement can hang, terminate normally, or terminate abruptly (involving various possible abnormalities). Just unfolding the definitions describing all possible outcomes quickly leads to screens full of unreadable PVS code. This complexity is managed by using many small rewrite steps for all cases in expressions and statements from the prelude files (and by letting LOOP generate additional rewrite rules which are specific for the

[8]On the fastest machines at our disposal (a Pentium II 300 with 128M RAM, or an UltraSPARC 2 (model 2200) with 1000M RAM admitting maximally 1 CPU per user) this takes in interactive mode (with prover output to the screen, via emacs) about 2 min. run time, and a bit less than 3 min. real time (including garbage collecting). In batch mode, it takes less than half of the run time.

translated class), so that in principle, complete definitions never have to be expanded.

4.2 Inheritance: overriding, hiding and late binding

The previous example does not involve any typically object-oriented aspects. In this subsection we consider some examples with inheritance. At the level of interfaces (in PVS), inheritance is handled by nesting. For example, suppose we have classes:

```
class A { float r, s; }
class B extends A { float r, t; }
```

Then interface types are generated of the following form.

```
AIFace : TYPE = [#
                   r : float,
                   s : float,
                   r_becomes : [float -> Self],
                   s_becomes : [float -> Self],
                #]

BIFace : TYPE = [#
                   super_A : AIFace,
                   r : float,
                   t : float,
                   r_becomes : [float -> Self],
                   t_becomes : [float -> Self]
                 #]
```

Definitions `r(c)`, `s(c)`, `r_becomes(c)` and `s_becomes(c)` are then automatically generated for a coalgebra `c : [Self -> AIFace[Self]]` of class `A`, as discussed in the beginning of the previous section. For a coalgebra `c : [Self -> BIFace[Self]]` of class `B`, there are similar definitions, of the following form—where we concentrate on the fields.

```
r(c) : [Self -> float] =
  LAMBDA(x : Self) : r(c(x))
t(c) : [Self -> float] =
  LAMBDA(x : Self) : t(c(x))
s(c) : [Self -> float] =
  LAMBDA(x : Self) : s(super_A(c(x)))
A_r(c) : [Self -> float] =
  LAMBDA(x : Self) : r(super_A(c(x)))
```

In this manner the non-hidden field `s` from `A` is directly available in `B`. And the hidden field `r` is also available, but under a different name `A_r`. This mechanism avoids subtyping. It also underlies inheritance in CCSL [13].

At the level of method definitions we have to be careful, because of overriding and late binding. In a class `C` we repeat all method definitions from super classes of `C`, but instantiated with a coalgebra from `C`. Late binding then works automatically, but some care is needed to make sure that appropriate fields are selected.

As an example we consider the translation of the following series of Java classes `Parent` – `Child` – `GrandChild`, defined via inheritance.

```
class Parent {
  int i;
  void base() { i = 4; }
}

class Child extends Parent {
  int i, j;
  void deriv() { j = 1; base(); }
}

class GrandChild extends Child {
  void base() { i = 8; }
}
```

The declaration `int i` in `Child` "hides" the `i` from `Parent`, see [10, Section 8.3], but running `deriv` in `Child` will affect `i` in `Parent`, and not `i` in `Child`. In contrast, running `deriv` in `GrandChild` will affect `i` in `Child`, but not `i` in `Parent`, due to the late binding mechanism which determines that within the `GrandChild` class `deriv` will call the (redefined) `base` method from `GrandChild`.

The aim is to prove the right values of the `i`'s and `j` after running `deriv` in `Child` and in `GrandChild`, via automatic rewriting. The difficulty in this example is not located in the complexities of the expressions involved, but in getting the bindings right. The LOOP tool solves this difficulty by suitably repeating method definitions from superclasses in subclasses.

Using the LOOP translation of these Java classes into PVS we first show that the method `deriv` terminates normally (and does not hang or terminate abruptly). Then we can express the values of the fields in the resulting state after `deriv` in terms of the original values as follows. For a `Child` coalgebra `d(p)` acting on an arbitrary memory position `p` this is expressed in the following result.

```
Child_deriv : LEMMA
  norm??(deriv(d(p))(x))
    AND
  i(d(p))(ns?(deriv(d(p))(x))) = i(d(p))(x)
    AND
  j(d(p))(ns?(deriv(d(p))(x))) = 1
    AND
  Parent_i(d(p))(ns?(deriv(d(p))(x))) = 4
```

The first assertion in the conjunction states that running the method `deriv(d(p))` in an arbitrary state `x` terminates normally. The next three statements describe the values of the variables `i(d(p))`, `j(d(p))` and `Parent_i(d(p))` (*i.e.* `i` from the super class `Parent` of child coalgebra `d(p)`) when evaluated in the normal state (accessed by `ns?`) resulting from running `deriv(d(p))`.

For a `GrandChild` coalgebra `gc(p)` the required result is:

```
GrandChild_deriv : LEMMA
  norm??(deriv(gc(p))(x))
    AND
  i(gc(p))(ns?(deriv(gc(p))(x))) = 8
    AND
  j(gc(p))(ns?(deriv(gc(p))(x))) = 1
    AND
  Parent_i(gc(p))(ns?(deriv(gc(p))(x))) =
    Parent_i(gc(p))(x)
```

Both lemmas are proved by automatic rewriting[9].

[9] The `Child_deriv` lemma requires 36 rewrite steps, taking about 10 sec. run time, and the `GrandChild_deriv` lemma is proved in 40 steps, again in 10 sec.

4.3 Local variables and recursion

Sections 3 and 4.1 already briefly discussed how the LOOP tool handles parameters, local variables, and special variables for returns. Here we describe this in more detail, in the context of a recursive definition of the factorial function:

```
class Fac {
  int fac (int n) {
    int i = 1;
    if (n != 0) { i = n * fac (n - 1); }
    return i;
  }
}
```

The method `fac` has a parameter `n`, a local variable `i` and it returns a value of type `int`. Local variables, just like parameters are handled in the LOOP translation via `LET`s in PVS, using a fresh memory location (see Figure 2). This will guarantee that whenever a new block is started (which may happen frequently, like in the above recursive method) new storage capacity is available.

Recursion is handled by explicitly calculating the least fixed point of a suitable functional. This is done by determining if there is a number n so that after n iterations the method does not hang. If this is the case the recursive definition is unfolded, the least such n times; otherwise the method hangs. The least fixed point of this functional provides the semantics for the recursive definition. The fixed point approach actually works for mutually recursive functions as well, because we do not iterate individual methods but the whole coalgebra (incorporating all methods). Details will be described elsewhere.

Using this formalisation we can prove,

```
fac_lem : LEMMA
  FORALL(x : OM?, n : int_java) :
    IF n >= 0
    THEN norm??(fac(c(p))(x, n)) AND
         res?(fac(c(p))(x, n)) = pvs_fac(n)
    ELSE hang??(fac(c(p))(x, n))
    ENDIF
```

where `pvs_fac` is the usual factorial function, defined in PVS (for positive numbers). The proof proceeds by reasoning about iterations (using induction over the natural numbers).

4.4 A while loop, with break and continue

Towards the end of Section 2 the semantics of a while statement is sketched: first it is decided if/when the loop terminates. If not, the while statement hangs, otherwise it comes down to executing the body the appropriate number of times. In Java, a while statement can terminate for two reasons: at some stage (1) its condition evaluates to false, or (2) execution of its expression or body statement terminates abnormally, because of an exception, break or return. Here we consider the following simple example.

```
class Loop {
  int break_loop (int i) {
    lab : while (true) {
            if (i < 20){i++; continue lab;}
            else break;
          }
    return i;
  }
}
```

It involves a while loop in Java which terminates because of a break. Reasoning about while loops generally involves (see *e.g.* [3, 11, 9, 4]) a loop invariant and also a variant. The latter is a function from `Self` to some well-founded set, which decreases with every execution of the loop body. Variants are used for proving termination. Using this approach we can prove the following.

```
res?_break_loop : LEMMA
  norm??(break_loop(c(p))(x, i)) AND
  res?(break_loop(c(p))(x, i)) = max(i, 20)
```

More details of reasoning about such while loops with abnormalities will appear in [16].

4.5 An invariance result

So far we have only seen examples of user statements about individual methods in a Java class. The next two examples will consider a class as a whole, first in showing that a certain predicate is an invariant of a class, and second in showing that a class can be a model (or implementation) of a specification.

As mentioned briefly in the introduction, the LOOP tool not only translates Java classes into PVS, but also generates for each class appropriate notions of invariant and bisimulation. This involves some basic constructions from the theory of coalgebras (see [18]), which are ultimately based on ideas in categorical logic (see [14]). Here we concentrate on invariants. These are predicates on the state space, which, once they are true for a state `x`, will remain true no matter which public[10] methods (or assignments for public variables) are applied to `x`. The definition of invariance is different for every class, because it depends, for example, on the method types.

Consider for example the following Java class, describing a simple counter modulo `max`.

```
class Counter {
  private int max;
  private int val;
  int maximum() { return max; }
  int value() { return val; }
  void next() {
    if ( val < max ) { val = val + 1; }
    else { val = 0; } }
  void clear() { val = 0; }
  Counter(int n) { max = n; }
}
```

An invariant for this class is a predicate which is closed under application of `maximum`, `value`, `next` and `clear`—but not under assignments for the private variables `max` and `val`. Intuitively it is clear that the following predicate on the global memory `OM?` is an invariant.

[10] In Java there are many visibility modifiers, see [10, Section 6.6] many of which are related to Java's package system, but the LOOP tool only has `public` and `private`. The LOOP translation sends `private` in Java to `private`, and everything else to `public`. Within the LOOP tool, these visibility modifiers are (currently) only relevant for the notions of invariant and bisimulation.

```
BEGIN CCSLcounter : CLASSSPEC
  METHOD
    max    : Self -> int;
    val    : Self -> int;
    next   : Self -> Self;
    clear  : Self -> Self;
  ASSERTION
    max_next   :
      PVS max(next(x)) = max(x) ENDPVS
    max_clear  :
      PVS max(clear(x)) = max(x) ENDPVS
    val_next   :
      PVS val(next(x)) = IF val(x) < max(x)
                         THEN val(x) + 1
                         ELSE 0
                         ENDIF ENDPVS
    val_clear  :
      PVS val(clear(x)) = 0 ENDPVS
  CONSTRUCTOR
    new    : int -> Self;
  CREATION
    max_new    :
      PVS FORALL(n : int) : max(new(n)) = n ENDPVS
    val_new    :
      PVS FORALL(n : int) : val(new(n)) = 0 ENDPVS
END CCSLcounter
```

Figure 3: A counter class specification in CCSL

```
  val_below_max(d, p) : [OM? -> bool] =
    LAMBDA(x : OM?) :  0 <= max(d(p))(x) AND
                       0 <= val(d(p))(x) AND
                       val(d(p))(x) <= max(d(p))(x)
```

Proving this formally amounts to proving the next lemma,

```
  val_below_max_inv : LEMMA
    invariant?(d(p))(val_below_max(d, p))
```

in which `invariant?` is a predicate which is generated by the LOOP tool. It is not hard to prove this result, since most of the work is done via automatic rewriting.

4.6 A Java implementation satisfying a CCSL class specification

The introduction of this paper describes how the LOOP tool accepts and translates both class specifications (in a language called CCSL, see [13]) and class implementations (in Java) as input. These translations can be combined, so that one can check in PVS whether a Java class implements a CCSL specification. We briefly illustrate this combination by (re)considering the Java counter class from the previous subsection. A specification of such a counter (modulo `max`) is presented in Figure 3. It is written in CCSL [13], and this language is hopefully self-explanatory.

We concentrate on the (validity of the) assertions[11]. The LOOP tool translates the CCSL counter specification into a series of PVS theories. In one of these theories, the assertions in Figure 3 are combined into a single predicate `CCSLcounterAssert?` on a CCSLcounter coalgebra

```
  c : [Self -> CCSLcounterIFace[Self]]
```

which combines the methods of the CCSL counter class in a single function. In order to show that the Java implementation forms a model of this CCSL specification we first have to transform a coalgebra describing the Java class into a coalgebra for this CCSL class, and then show that the assertions of the CCSL class are satisfied. In PVS these steps are as follows.

```
  p : VAR nat
  d : VAR [p : nat -> (CounterAssert?(p))]

  counter(d, p) : [OM? -> CCSLcounterIFace[OM?]] =
     LAMBDA(x : OM?) :
       (#
         max := res?(max(d(p))(x)),
         val := res?(val(d(p))(x)),
         next := ns?(next(d(p))(x)),
         clear := ns?(clear(d(p))(x))
       #)

  CCSLcounter_JavaImplementation : LEMMA
    CCSLcounterAssert?(counter(d, p))
```

The latter lemma establishes the desired implementation result. It is proved automatically by rewriting[12].

4.7 Component classes and casting

Classes can form components of other classes: if `MyClass` is already defined, then one can declare a field `MyClass mc` in some other class (or even in `MyClass` itself). Once `mc` is properly initialised, methods from `MyClass` can be applied to `mc`. But also, `mc` can be cast to superclasses of `MyClass`, see [5, Subsection 5.13.2] or [10, Section 5.5]. This creates substantial difficulties for the translation to PVS, which we can currently only handle "by hand". That is, we know how to translate such casting, but LOOP does not[13].

Casting in Java introduces a difference between fields and methods (see [5, Section 3.4]): suppose `B` is a subclass of `A`, and both `A` and `B` have a field `f` and a method `m` (of the same type). Thus `f` from `A` is "hidden" in `B` and `m` from `A` is "overridden" in `B`. Let `b` be of type `B`, and consider its cast `a = (A)b` to `A`. Then `a.f` is `f` in `A`, whereas `a.m` is `m` in `B`. This difference is highly relevant for reasoning about casting[14].

[11] In principle, the creation conditions for constructors are handled similarly.

[12] The definition of the `counter` function also generates several obligations ("tcc's") to prove that the Java methods terminate normally, so that their result `res?` or resulting normal state `ns?` can be accessed. Also these obligations are handled by automatic rewriting.

[13] The reason is that in order to perform the translation of a cast from class `A` to class `B` we need to know both `A` and `B`. This information can only be obtained by letting LOOP typecheck Java programs, because casting is often done implicitly. Java typechecking is currently being added to the LOOP tool.

[14] Our translation "by hand" handles this difference by letting `a` look at `b` with an adapted coalgebra. This can also be expressed in terms of "two references" to `a`, see [5, page 69]: "one reference as its actual class and the other as its superclass".

5 Conclusions and further work

We have sketched the essential ingredients of a (partial) translation of Java classes into the higher order logic of PVS, as performed by the LOOP tool. Also we have shown how this allows us to prove some elementary properties about Java classes in PVS. This may be seen as applied semantics of programming languages. Space restrictions prevent us from describing all details here, but more will be presented in future work.

It may be clear that this project is far from finished. We will continue to extend the translation to aspects of Java which are currently not covered. Being able to reason about threads is a long-term goal, which will first require a fundamental study of the semantics of threads in Java (see also [8]) within the coalgebraic approach underlying the LOOP tool. Another future extension is to define an appropriate "Annotated Java" language consisting of standard Java with correctness assertions added as comments. These should be translated into appropriate verification conditions. To conclude, we would like to emphasise that this is very much a research project and that major applications are not foreseen in the near future.

References

1. M. Abadi and L. Cardelli. *A Theory of Objects.* Monographs in Comp. Sci. Springer, 1996.
2. M. Abadi and K.R.M. Leino. A logic of object-oriented programs. In M. Bidoit and M. Dauchet, editors, *TAPSOFT'97: Theory and Practice of Software Development*, number 1214 in Lect. Notes Comp. Sci., pages 682–696. Springer, Berlin, 1997.
3. K.R. Apt. Ten years of Hoare's logic: A survey—part I. *ACM Trans. on Progr. Lang. and Systems*, 3(4):431–483, 1981.
4. K.R. Apt and E.-R. Olderog. *Verification of Sequential and Concurrent Programs.* Springer, 1991.
5. K. Arnold and J. Gosling. *The Java Programming Language.* Addison-Wesley, 2$^{\text{nd}}$ edition, 1997.
6. J.W. de Bakker. *Mathematical Theory of Program Correctness.* Prentice Hall, 1980.
7. V.A. Carreño and P.S. Miner. Specification of the IEEE-854 floating-point standard in HOL and PVS. In E.Th. Schubert, Ph.J. Windley, and J. Alves-Foss, editors, *Higher Order Logic Theorem Proving and Its Applications*, 1995. Category B Proceedings, available at URL `http://lal.cs.byu.edu/lal/hol95/Bprocs/indexB.html`.
8. P. Cenciarelli, A. Knapp, B. Reus, and M. Wirsing. From sequential to multi-threaded Java: An event-based operational semantics. In M. Johnson, editor, *Algebraic Methodology and Software Technology*, number 1349 in Lect. Notes Comp. Sci., pages 75–90. Springer, Berlin, 1997.
9. M.J.C. Gordon. Mechanizing programming logics in higher order logic. In G. Birtwistle and P.A. Subrahmanyam, editors, *Current Trends in Hardware Verification and Automated Theorem Proving*, Springer, 1989.
10. J. Gosling, B. Jay, and G. Steele. *The Java Language Specification.* Addison-Wesley, 1996.
11. D. Gries. *The Science of Programming.* Springer, 1981.
12. C.A. Gunter. *Semantics of Programming Languages. Structures and Techniques.* The MIT Press, Cambridge, MA, 1992.
13. U. Hensel, M. Huisman, B. Jacobs, and H. Tews. Reasoning about classes in object-oriented languages: Logical models and tools. In Ch. Hankin, editor, *European Symposium on Programming*, number 1381 in Lect. Notes Comp. Sci., pages 105–121. Springer, Berlin, 1998.
14. C. Hermida and B. Jacobs. Structural induction and coinduction in a fibrational setting. Inf. & Comp., to appear, 1998.
15. C.A.R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12:576–580, 583, 1969.
16. M. Huisman and B. Jacobs. Hoare logic with abrupt termination. In preparation, 1998.
17. B. Jacobs. Objects and classes, co-algebraically. In B. Freitag, C.B. Jones, C. Lengauer, and H.-J. Schek, editors, *Object-Orientation with Parallelism and Persistence*, pages 83–103. Kluwer Acad. Publ., 1996.
18. B. Jacobs. Invariants, bisimulations and the correctness of coalgebraic refinements. In M. Johnson, editor, *Algebraic Methodology and Software Technology*, number 1349 in Lect. Notes Comp. Sci., pages 276–291. Springer, Berlin, 1997.
19. B. Jacobs and J. Rutten. A tutorial on (co)algebras and (co)induction. *EATCS Bulletin*, 62:222–259, 1997.
20. X. Leroy. *The Objective Caml system, Documentation and user's guide; Release 1.07*, 1997. Available at URL `http://pauillac.inria.fr/ocaml/htmlman`.
21. J. Loeckx and K. Sieber. *The Foundations of Program Verification.* Wiley, 1987.
22. S. Owre, J.M. Rushby, N. Shankar, and F. von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Trans. on Softw. Eng.*, 21(2):107–125, 1995.
23. H. Reichel. An approach to object semantics based on terminal co-algebras. *Math. Struct. in Comp. Sci.*, 5:129–152, 1995.
24. D. Rémy and J. Vouillon. Objective ML: An effective object-oriented extension of ML. *Theory & Practice of Object Systems*, 1998, to appear.
25. J. Rutten. Universal coalgebra: a theory of systems. *Theor. Comp. Sci.*, 1998, to appear.
26. R.D. Tennent. *Semantics of Programming Languages.* Prentice Hall, 1991.