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

April 14, 1998

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 logic, and a back-end theorem prover (namely PVS, developed at SRI) for reasoning. In several examples we will demonstrate how non-trivial properties of Java programs and classes can be proved 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 [4, 8]. 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 [18]) 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 one's 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.

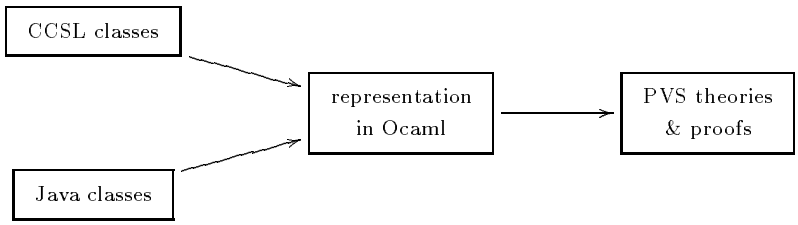
In brief, a proof assistant is like a "sceptical colleague" who patiently checks all details and is willing to do routine tasks.

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, 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 will 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.

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

What we shall describe is a part of a more general LOOP tool described in [11] for translating object-oriented specifications into higher-order logic. The tool performs the following transformations.



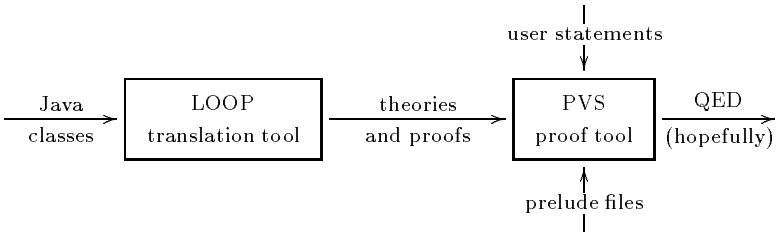
First it reads (lexes and parses) classes in CCSL (Coalgebraic Class Specification Language, see [11]) or Java, and transforms these in some internal representation in the Ocaml programming language [20, 16] (the implementation language of LOOP). This representation is subjected to certain internal analyses, *e.g.* for establishing the inheritance relationships between classes. Finally, it is transformed into theories (and proofs) of the PVS theorem prover [18]. Much of the internal code of the tool is shared for both the translations from CCSL and Java classes to PVS.

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 in a file `MyClass.java`. The tool then produces a new file[3], called `MyClass_basic.pvs`. It contains a translation into the higher-order logic of PVS of all of the Java classes in the original file `MyClass.java`. 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. The statements that the user wishes to prove about the (translated) Java class should be put here. All these (`.pvs`) files can be loaded into the PVS theorem prover, and the user can start trying to prove the desired results, using the translation of the LOOP tool.

[2] We always assume that Java classes which are fed into the LOOP tool are accepted by a (standard) Java compiler.

[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.

Summarising:



In this paper we shall 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 interesting details of the resulting translation, (3) some propositions 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.* [5, 10]. 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 $[\![-]\!] : \mathcal{L} \to \mathcal{D}$ 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 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: what is a subclass

of what, and aggregation: what is a component of what) 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.* [19, 13, 14].

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

What distinguishes the current project from other formal approach 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 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 component classes.

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

Figure 1: The four main datatypes used for the translation

2 Java semantics in the higher-order logic of PVS

In this section we will 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 over 200K (about 7000 lines). We will 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 tranlation is based because `?` 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 is probably best simply to ignore all these `?`'s. A more important point is that these state transformer functions are examples of coalgebras (see [15]): 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 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 two codomain types of statements and expressions are abbreviations:

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

involving four data types, see Figure 1. The 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 *etc*. 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)`. 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 result in the output type `Out` of the expression, together with a (new) state, because expressions can have side-effects. 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 (exceptions, returns, breaks and continues). Similarly, 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

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

`[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)(x) = 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 way we define a conditional statement `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)(x) = 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?(E2SAbn?(dev?(c(x))))
                           ENDIF
```

(The `E2SAbn?` term turns an expression abnormality into a statement abnormality.)

In this manner, all Java constructs are translated in the prelude files, following the explanations in [8]. Another such example is a `RETURN` statement in PVS, defined as:

```
RETURN(x) = abnorm?(rtrn?(x))
```

It creates a return abnormality, see Figure 1. 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)(x) =
  IF norm??(e(x))
  THEN
    LET r = res?(e(x)),
        y = ns?(e(x)) IN
    IF norm??(d(y))
    THEN norm?(ns?(d(y)),
               r AND res?(d(y)))
    ELSE d(y)
```

```
(e ANDTHEN d)(x) =
  IF norm??(e(x))
  THEN
    LET r = res?(e(x)),
        y = ns?(e(x)) IN
    IF NOT r
    THEN e(x)
    ELSE d(y)
    ENDIF
```

```
  ENDIF                                ELSE e(x)
ELSE e(x)                              ENDIF
ENDIF
```

Notice how side-effects are propagated through these composite expressions. 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.

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.

In one of the prelude files a model `GM?` is defined of a global memory, containing an infinite[5] number of memory cells. 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 this global state space `GM?`.

3 Translating Java classes

The LOOP tool calculates a function $[\![-]\!]$ which assigns meaning to Java classes. It follows the Java grammar [8, Chapter 19], and takes for example $[\![$`e1`$]\!]$`AND`$[\![$`e2`$]\!]$ as PVS translation $[\![$`e1 & e2`$]\!]$ 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 an interface type (like in [11]). 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 basically give rise to the following interface (record) type in PVS.

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

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

Some additional variables may be incorporated in such an interface type of a class: local variables, parameter variables of methods, and return variables of non-void methods (occurring in this class). These variables are thus made global, but this is harmless. Name clashes are avoided by putting a local variable `i` in the interface as `loc?i`. Similarly, we use `par?j` for a parameter `j`, and `ret?meth` for the return variable of method `meth`, see for example Figure 3 in Subsection 4.1 below.

A coalgebra for class `MyClass`, say, in this context is a function in PVS of the form

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

where `MyClassIFace[Self]` is the interface type generated for class `MyClass` (like above). 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 [15].

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, db : int) :
  move(c?)(x?, da, db) = (
    E2S(fst_becomes(c?)(fst(c?) + da)) #
    E2S(snd_becomes(c?)(snd(c?) + db))
  )(x?)
```

(An aside about the translation: assignments are defined as expressions. When they are used as statements, like in this `move` method, an additional function `E2S` is inserted in the translation of the method body, which transforms an expression into a statement—basically by forgetting the output when the expression terminates normally, see Figure 1.)

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 [11].

This account simplifies matters slightly for explainatory purposes. We have already mentioned (at the end of the previous section) that all coalgebras operate on the same state space `GM?`, describing a global memory. Each field declaration, say `int i`, is implemented as a function which can read a value at a particular location in `GM?`. Similarly, the associated assignment operation (`i_becomes`) writes at this same location. This memory location is a PVS variable 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 `GM?` together with a coalgebra (of the class) acting on this position `p?`.

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, many statements and expressions have already been translated. 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 the former are approximations of real numbers (described precisely in the IEEE 754 floating point format). In order to translate accurately, 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.* [6]. 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

```
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;
    }
}
```

Figure 2: A class in Java with a weird method

```
FORALL (j : int_java) :
  lets_calculate(c?)(x?, j) =
    CATCH_EXPR_RETURN[GM?,int_java](
      E2S(par?j_becomes(c?)(const[GM?, int_java](j))) #
      (TRY_CATCH(
        (E2S(i_becomes(c?)((i(c?) *
               QUESTION(((i(c?) > const[GM?, int_java](5))))
                       ((((i(c?) ## i_becomes(c?)(inc(i(c?)))) //
                           par?j_becomes(c?)(dec(par?j(c?))))))
                       (const[GM?, int_java](8)))))))
        ((:
          (Exception,
          (E2S((par?j(c?) ## par?j_becomes(c?)(dec(par?j(c?))))))) #
          (E2S(ret?lets_calculate_becomes(c?)((i(c?) - par?j(c?)))) #
          RETURN))
        :))) #
      (E2S(ret?lets_calculate_becomes(c?)((i(c?) + par?j(c?)))) #
      RETURN))
    (ret?lets_calculate(c?))
    (x?)
```

Figure 3: The LOOP translation of the weird method in PVS

an exception object (like the `e` in Figure 2) 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 will elaborate some examples. In particular, we will 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.

4.1 A weird method

Consider the Java class in Figure 2. 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 takes the value of `i` as first argument, then `i` is incremented, then `j` is decremented, and the resulting value (if `j`) is taken as second argument, so that the remainder can finally be computed.
- Exception handling: if `j = 1`, then the remainder operation (translated as `//`) will throw 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 using 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 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 `lets_calculate` method given in Figure 3. This translation is probably unreadable, and not really meant for 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 `?` 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 the left hand side `lets_calculate(c?)(x?, j)` 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.

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

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
```

Figure 4: A lemma in PVS about the weird expression method

- A special variable `ret?lets_calculate` (together with an associated assignment) is used for the 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)[6].

- 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. This `##` operation returns the result of its first argument (and ignores the second result) together with the state obtained from running the second argument on the state resulting from the first argument.

- The `TRY_CATCH` statement takes as argument a list—indicated in PVS by `(: ··· :)`—of pairs 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 is described in Figure 4. 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 ouput 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

[6] Using such an auxiliary variable also ensures that parameters are passed by value, see [4, 2.6.1].

```
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; }
}
```

Figure 5: Late binding example in Java

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 lemma in Figure 4 can be proved in PVS by using basically only two proof commands: (load-rewrite-theories $\cdots$) and (do-rewrite). All the expressions in the lemma are then suitably rewritten (following the evaluation strategy of Java) to the required result. This involves 222 single rewrite steps[7]. 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 abnormally (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 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 will consider the translation of the series of JAVA classes `Parent` – `Child` – `GrandChild` in Figure 5, defined via inheritance. The declaration `int i` in `Child` "hides" the `i` from `Parent`, see [8, Section 8.3], but running `deriv` in `Child` will affect `i` in `Parent`, and not `i` in `Child`. In contrast, running

[7] 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.

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

Figure 6: A recursive factorial function in Java

`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. This is achieved in the LOOP tool by suitably repeating method definitions from superclasses in subclasses.

In the LOOP translation of these JAVA classes into PVS we first have to 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 `deriv(d?(p?))` in an arbitrary state `x?` is normal (*i.e.* 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
```

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

Figure 7: An example of a while loop in Java, using break and continue

```
  AND
 Parent_i(gc?(p?))(ns?(deriv(gc?(p?))(x?))) = Parent_i(gc?(p?))(x?)
```

Both lemmas are proved by automatic rewriting[8].

4.3 Local variables, and recursion

In Sections 3 and 4.1 it was already briefly discussed how the LOOP tool handles parameters, local variables and special variables for returns. Here we will describe this in more detail, in the context of a recursive definition of the factorial function (see Figure 6).

Function `fac` has a parameter `n`, a local variable `i` and it returns a value of type `int`. As explained, local variables, parameters and return variables are made global, and potential name clashes with any identifier from the Java source are avoided by naming them `loc?i`, `par?n` and `ret?fac`, respectively.

Upon entry of each recursive call, "new" variables `par?n` and `loc?i` have to be available, and they must be discarded after leaving this call. This is realised using a `BLOCK` statement, with two parameters: a (composite) statement and a restore function. When such a `BLOCK` is executed in state `x`, first the statement runs on `x`, say resulting in a state `y`. Then the local variables are restored to their values (or references) from `x`, yielding a state `z`. The `BLOCK` statement then returns `z`.

Now, one can prove properties about the factorial function, such as termination for all `n`, simply by induction on `n`. Also, the facts that it returns what is expected can be proven. In these proofs, one has to be careful: automatically rewriting `fac` to its body loops.

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

[8] 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.

```
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; }
}
```

Figure 8: A Counter class in JAVA

condition evaluates to false, or (2) execution of its expression or body statement terminates abnormally, because of an exception, break or return. More details about reasoning about such while loops will appear elsewhere.

Figure 7 shows an example of class with a while loop in Java—which terminates because of a break. After translating this class with LOOP, we can prove that its method `break_loop` terminates after `max(21 - i, 1)` iterations (where `i` is the actual parameter of method `break_loop`). Also, we can prove that the value of the parameter `par?i` will be `max(20, i)`, after termination of the `WHILE` statement.

```
par?i_WHILE : LEMMA
  FORALL (i : int) : par?i[GM?](d?(p?))(x?) = i IMPLIES
    par?i(d?(p?))(bs?(dev?((WHILE_DO(up?("lab"))
                  (const[GM?, bool](true))
                  (IF_THEN_ELSE
                    (par?i(d?(p?)) < const[GM?, int_java](20))
                    ((E2S[GM?, int_java](par?i(d?(p?))
                         ## par?i_becomes(d?(p?))(inc(par?i(d?(p?))))))
                         # (CONTINUE("lab")))
                    (BREAK)))(x?)))) = max(20, i)
```

Reasoning about such while programs generally follows standard approaches, see *e.g.* [9, 17, 3]. We plan to incorporate this via suitable proof methods in PVS.

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 [14]), which are ultimately based on ideas in categorical logic (see [12]). Here we will 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[9] methods (or assignments for public variables) are applied to `x`. Consider for example the class in Figure 8, describing a simple counter modulo `max`. 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 `GM?` is an invariant.

```
val_below_max(d?, p?) : [GM? -> bool] =
  LAMBDA(x : GM?) :  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 following 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 both class specifications (in a language called CCSL, see [11]) and class implementations (in Java) as input. An obvious question arises: can one formulate a class specification in CCSL, and a class implementation in Java, and then show that the (translated) Java class forms a model (or implementation) of the (translated) CCSL class specification. The answer is yes. We shall briefly indicate how this is done, by (re)considering the Java counter class in Figure 8. A specification of such a counter (modulo `max`) is presented in Figure 9. It is written in CCSL [11], and this language is hopefully self-explanatory.

We shall concentrate on the (validity of the) assertions[10]. The LOOP tool translates the CCSL counter specification into a series of PVS theories. In one of these theories, the assertions in Figure 9 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

[9] In Java there are many visibility modifiers, see [8, 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.

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

```
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 9: A counter class specification in 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?) : [GM? -> CCSLcounterIFace[GM?]] =
     LAMBDA(x? : GM?) :
       (#
         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 is proved automatically by rewriting[11]. This establishes the desired implementation result.

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`

[11] 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.

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 [4, Subsection 5.13.2] or [8, 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[12].

Casting in Java introduces a difference between fields and methods (see [4, 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[13].

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 programs 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 [7]) within the coalgebraic approach underlying the LOOP tool. 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 and E.-R. Olderog. *Verification of Sequential and Concurrent Programs.* Springer, 1991.

[12] 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. And Java typechecking is not incorporated in LOOP yet.

[13] 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 [4, page 69]: "one reference as its actual class and the other as its superclass".

[4] K. Arnold and J. Gosling. *The Java Programming Language*. Addison-Wesley, 2^{nd} edition, 1997.

[5] J.W. de Bakker and E. Vink. *Control Flow Semantics*. The MIT Press, Cambridge, MA, 1996.

[6] 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`.

[7] 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.

[8] J. Gosling, B. Jay, and G. Steele. *The Java Language Specification*. Addison-Wesley, 1996.

[9] D. Gries. *The Science of Programming*. Springer, 1981.

[10] C.A. Gunter. *Semantics of Programming Languages. Structures and Techniques*. The MIT Press, Cambridge, MA, 1992.

[11] 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.

[12] C. Hermida and B. Jacobs. Structural induction and coinduction in a fibrational setting. Inf. & Comp., to appear, 1998.

[13] 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.

[14] 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.

[15] B. Jacobs and J. Rutten. A tutorial on (co)algebras and (co)induction. *EATCS Bulletin*, 62:222–259, 1997.

[16] X. Leroy. *The Objective Caml system, Documentation and user's guide; Release 1.05*, 1997. Available at URL `http://pauillac.inria.fr/ocaml/htmlman`.

[17] J. Loeckx and K. Sieber. *The Foundations of Program Verification*. Wiley, 1987.

[18] 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.

[19] H. Reichel. An approach to object semantics based on terminal co-algebras. *Math. Struct. in Comp. Sci.*, 5:129–152, 1995.

[20] D. Rémy and J. Vouillon. Objective ML: An effective object-oriented extension of ML. *Theory & Practice of Object Systems*, 1998, to appear.