The LOOP compiler for Java and JML

Joachim van den Berg, Bart Jacobs

Computing Science Institute, University of Nijmegen
Toernooiveld 1, 6525 ED Nijmegen, The Netherlands
{joachim,bart}@cs.kun.nl

Abstract This paper describes the architecture of the LOOP tool, which is used for reasoning about sequential Java. The LOOP tool translates Java and JML (a specification language tailored to Java) classes into their semantics in higher order logic. It serves as a front-end to a theorem prover in which the actual verification of the desired properties takes place. Also, the paper discusses issues related to logical theory generation.

1 Introduction

Being able to verify programs has always been a major topic in computer science. For this purpose many artificial, mathematically clean, programming languages have been introduced, since reasoning about real, dirty, programming languages is far from easy. Due to the progress in the field of theorem proving, and the increase in computing power, it has become feasible now to reason about real programming languages. Also, specialised tools—like the LOOP tool—contribute to this feasibility.

Using theorem provers for program verification becomes more and more common. There are numerous advantages to the use of theorem provers for doing proofs over doing proofs by hand: theorem provers are very precise, they can do lots of, often boring, proof steps in a few seconds, they keep track of the list of proof obligations which are still open, and do a lot of bureaucratic administration for the user. This is especially relevant in the area of program verification where usually many cases have to be distinguished and the proofs themselves are not so difficult (in comparison to mathematics).

Since Java is one of the most popular programming languages around, it is also of particular interest for researchers. Many research groups are focusing on specification and verification of Java programs at source-code level, using various tools, *e.g.*

- ESC/Java [23] is an extended static checker for Java (including threads), which can detect certain runtime errors at compile time, by using a built-in theorem prover. By using this checker, many (but not all) errors can be found without user interaction. ESC/Java uses a specification language which has recently been integrated with JML [15].

- Jive [17] is a verification environment in which a user can write Java source-code as well as its specification. It is connected with a theorem prover, currently this is PVS [19], which is used to verify proof obligations. Jive's user interface takes care of the interaction with PVS. With Jive, one is currently able to reason about the sequential kernel of Java, but not about exceptions, a crucial part of Java.
- In the Bali project a deep embedding of a semantics for Bali, a Java subset, in Isabelle [20] has been developed, with various meta-theoretical results: formalisation of the type system to prove type-safety [18], soundness and completeness of an appropriate Hoare logic. This project is not primarily focussed on verification of concrete programs.
- The KeY project [1] aims at integrating formal specification and verification tools into the software engineering process. Within this project a dynamic logic for JavaCard, Java's subset for smart card programming, has been developed. The verification tool for this project is still under development.
- The Bandera project [5] extracts a non-finite-state model from Java source-code, and applies program analysis, abstraction and transformation techniques to it, in order to get a finite-state model. This model is abstractly represented, enabling the generation of concrete models for various model checking tools. The tools developed in this project are applied to several Java case studies.

The LOOP project [21] focuses on specification and verification of sequential Java. For this part of Java a formal semantics has been developed, based on coalgebras. JML is the language used to specify Java classes. For the kernel part of JML—invariants, behaviour specifications, including modifiable clauses—a formal semantics is being developed.

Within the LOOP project a special purpose compiler, the LOOP tool, has been built which incorporate these semantics of Java and JML. The output of the LOOP tool is a series of logical theories for the theorem provers PVS and Isabelle. This gives the verifier a choice of proof tool. Typically, when a user wants to reason about a Java class, (s)he uses the LOOP tool for the translation, and reasons about the program in the language semantics using a theorem prover. The LOOP approach makes use of existing, general purpose theorem provers, and concentrates on building a dedicated front-end for a particular application area, because developing a (dedicated) theorem prover is a project on its own. Reasoning goes via a combination of applying semantic-based Hoare logic rules and automatic rewriting. Several papers about the underlying semantics and logic have already been published [14,3,9,10,12,13]. This paper focuses on the tool itself.

Automatic translation of Java classes into a series of logical theories has several advantages above manual translation. The LOOP translation process is, boring, error-prone, and time consuming. A translated Java class is usually much larger in size than the original. A tool will do such a translation within a few seconds, without complaining, and without errors (if the translation function is implemented correctly). Another advantage is that with tool support the generated theories can be fine-tuned to achieve more efficiency in proofs, which is hardly possible when generating theories by hand.

In comparison to the projects mentioned above there are the following distinguishing features of the LOOP project.

- The ESC/Java tool involves no user interaction, is fast and easy to use, but can only detect a limited class of errors. With the LOOP tool the user has to engage in interactive program verification, using the back-end proof tool, but there are no inherent limitations to what can be (dis)proved. Thus, the ESC/Java and LOOP tools are complementary and can very well be used in combination, especially because they use the same specification language (namely JML).
- The Jive approach is closest to the LOOP approach. It differs however in its syntax-based approach, via a dedicated user interface, allowing reasoning about the actual program text (and not about its meaning). The specification language of the Jive tool resembles JML. It is too early to judge and compare these two approaches in actual examples.
- The Bandera project aims at verification of Java programs (especially involving threads) using model checkers. Similar to the LOOP project, output is generated for back-end tools that do the actual verification. However, model checkers instead of theorem provers are used. A general problem with multi-threaded Java is that the level of granularity is not well-defined.
- The Bali and KeY projects have not been used (yet) on substantial concrete examples of Java programs, making a comparison premature.

This paper is organised as follows. Section 2 describes the modular architecture of the LOOP tool. Section 3 describes some issues related to the theory generation. Section 4 briefly describes how to use the LOOP tool, and finally Section 5 gives an overview of possible application areas.

2 The architecture of the LOOP tool

As shown in Figure 1, the LOOP tool accepts three languages with object-oriented features, namely CCSL, Java, and JML. It serves as a front-end for a theorem prover which, in this figure, is PVS. The LOOP tool can also serve as a front-end for Isabelle. The theorem prover is used to actually prove properties about the classes in the input languages, on the basis of the logical theories generated by the LOOP tool.

2.1 Input languages

Historically, the first input language is CCSL [7,22], short for Coalgebraic Class Specification Language. It is an experimental specification language, which is jointly developed at the University of Dresden and the University of Nijmegen. With this language one can write class specifications in an object-oriented way,

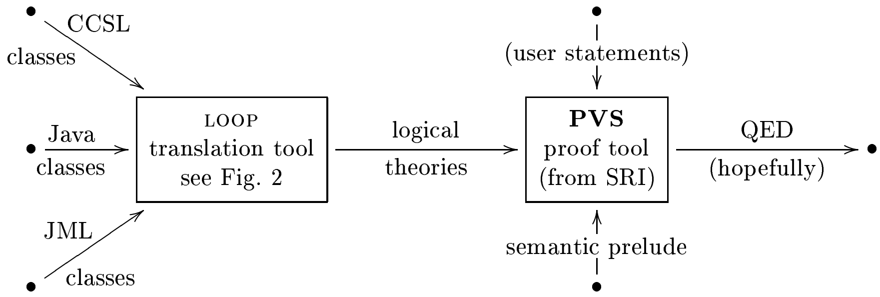


Figure1. Overview of the LOOP project

i.e. one can write specifications with attributes, methods, and constructors. It also supports inheritance and subtyping. CCSL uses a coalgebraic semantics for classes and supports tailor-made modal operators for reasoning about class specifications. In this paper we concentrate on the input languages Java and JML, and refer to [7,22] for more information on CCSL.

The second input language is Java—one of the most popular object-oriented programming languages. Our semantics for sequential Java, *i.e.* Java without threads, closely follows the Java Language Specification (JLS) [6]. More information about this semantics can be found in [14,3,9,10,12].

The third input language is JML, short for Java Modeling Language. JML is a behavourial interface specification language, tailored to Java, and primarily developed at Iowa State University. It is designed to be easy to use for programmers with limited experience in logic. Therefore, it extends Java such that a user can write (class) invariants, and pre- and post-conditions for methods and constructors within the source code, making use of Java expressions (extended with various logical operators) to formulate the desired properties. All extensions of JML are enclosed between Java's comment markers, and will therefore not influence the program's behaviour. A typical JML specification for a method `m` looks as follows.

```
/*@ behavior
  @    requires : <precondition>
  @  modifiable : <fields>
  @     ensures : <postcondition>     // when terminating normally
  @     signals : (E) <postcondition> // when terminating abruptly
  @                                   // because of exception E
  @*/
  void m () { ... }
```

2.2 LOOP tool internals

In Figure 2 the view on the LOOP tool is enlarged. Here a view is considered where the tool accepts Java classes (and interfaces)[1]. The first three passes can be viewed as the first part of a standard Java compiler.

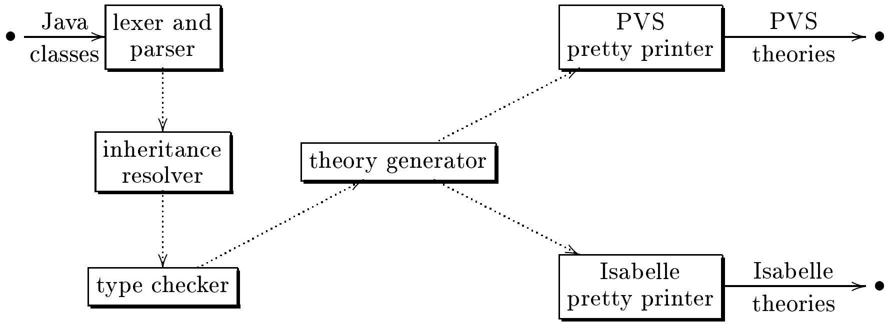


Figure2. The "exploded" view on the LOOP tool

Standard techniques are used to build a lexer and parser, following the definition of the Java syntax in the JLS. During parsing, unknown types—class and interface types—are not resolved. These types are stored as (tagged) strings in the abstract syntax tree, and resolved in a later pass.

The inheritance resolver establishes relations between classes by resolving the unknown types. Also, in this pass overridden methods and hidden fields in Java are internally marked as overridden and hidden.

The type checker computes the type of every expression occurring in the input classes. A type checker is needed, since the overloading mechanism of Java is more powerful than the ones of PVS and Isabelle. Therefore, definitions in PVS and Isabelle are often provided with explicit types.

At this point a standard Java compiler would generate a bytecode file for each class. Instead, the LOOP tool translates each Java class into its semantics, in the form of a series of logical theories. These theories are produced internally in an abstract way using abstract logic syntax (ALS), see Subsection 3.3 below.

Finally, to come to concrete theories, a last pass, a pretty printer, is implemented to translate the ALS into concrete logic syntax. Abstract theories provide a powerful technique to produce concrete theories for different theorem provers[2]. Implementing such a pass is fairly simple. We have implemented two of these, one for PVS and one for Isabelle.

[1] In this paper 'Java class' may also be read as 'Java interface'. If not, it will explicitly be mentioned.

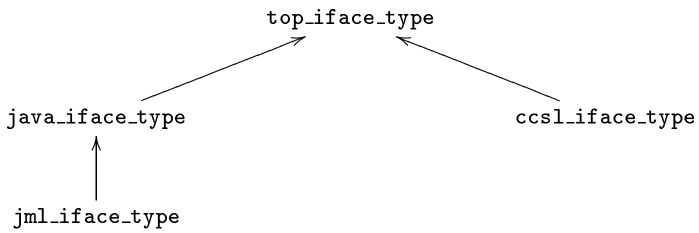
[2] The ALS involves standard constructions from higher order logic. Thus, it is in principle easy to generate output also for any theorem prover that provides (at least) higher order logic, *e.g.* COQ [2].

For JML the LOOP tool works similarly. Since JML is an extension of Java, the grammar of Java is extended, and for the logic of JML, that is based upon Java expressions, also the type checker is extended. The theories for the specifications are also abstractly generated. Notably, the pretty printer components of the LOOP tool are shared with the three input languages—CCSL, Java, and JML.

2.3 Implementation details

The OCaml language [16] is used to implement the LOOP tool. It comprises a number of tools, such as lex and yacc, a debugger, and a (native-code) compiler. OCaml is an ML dialect, supporting object-oriented features. It is a strongly typed (functional) programming language, *i.e.* every expression has a type which is automatically determined by the compiler. One great advantage of using a strongly typed language is that many potential program errors are caught by the compiler as type errors. The penalty for this is that one has to set up appropriately structured types first. This forms a non-trivial part of the implementation of the LOOP tool.

Internally, a Java/JML class and its members (fields, methods, and constructors) occurring in the input are stored as instances of certain OCaml classes. As root classes, we use two OCaml class types, `top_iface_type` for CCSL/Java/JML classes, and `top_member_type` for CCSL/Java/JML members. The "`top_`" class types contain common information, such as the name of the class, and the fields and methods defined in it. For each input language we introduce specialised class types, to deal with language specific properties.



Similarly, `top_member_type` has specialised class types for CCSL, Java, and JML members. Every "`_iface_type`" class type is mutually recursive with its "`_member_type`" variant. These types have a non-trivial structure, involving subtyping and mutual recursion in various forms.

Due to the object-oriented nature of the LOOP tool, it is easy to adapt the theory generation for the different input languages. Each "`_iface_type`" class type has a method that invokes the theory generation, which is overridden in specialised types.

Some non-technical details: the LOOP tool currently consists of over 58,000 lines of OCaml code (including documentation) of which 25,000 lines are used to implement the Java part, and 8,000 lines are used to extend it to JML. To implement CCSL 12,000 lines are used, and 13,000 lines of code are shared. Work on the LOOP tool started in 1997, and continues until this moment.

3 Generated theories

This section focuses on some typical issues and problems related to theory generation. The contents of the theories themselves are too complicated to describe here in detail, and are not directly relevant. See [9,8] for more information.

3.1 Mutually recursive classes and circular theories

The LOOP tool translates each Java (and JML) class into its semantics in higher order logic as a series of logical theories. It is not possible to generate this semantics as one single theory, since at several places in the source-code references to other classes might occur. Having such references might lead, in that case, to circular theories, via importings. This is not allowed in PVS and Isabelle. Hence, they have to be disentangled.

In Java source-code, references to other classes can occur at three places:

1. at inheritance level, but this does not lead to circularities, since a standard Java compiler detects if a class is a subclass of itself, *e.g.* `class A extends B` and `class B extends A` is illegal;
2. at interface level. The signatures of members of class `A` contain occurrences of class `B`, and vice versa;
3. at implementation level. In a method (or constructor) body in class `A` the class `B` occurs, *e.g.* via creating an object of class `B` or a field access of an object of type `B`, and vice versa.

For a concrete (toy) example of mutual recursion between Java classes, consider classes `A` and `B` in Figure 3, where the signature of method `m` in `A` has an occurrence of class `B`, and the signatures of both methods in `B` have occurrences of class `A`. Moreover, method `m` in `A` creates an object of class `B`, and method `n` assigns a value to a field of `b` (cast to `A`).

```
class A {
  int i;
  void m (B b) { b = new B(); }
}
```

```
class B extends A {
  A n() { return new A(); }
  void m (B b) { ((A)b).i = 1; }
}
```

Figure3. Mutually recursive Java classes

To prevent the generated theories from being circular, the semantics of each Java class is divided into three[3] tailor-made theories:

1. the *Prelude* theory defines a special type for objects and arrays of that class. This type can be the null reference or a reference pointing to a certain memory location where an object or a (multi-dimensional) array of objects of that class is stored.
2. the *Interface* theory defines the types of fields, methods, and possibly the constructors of that class. There is also a reference to the direct superclass, and superinterfaces, if any.
3. the *Bodies and rewrites* theory gives semantics to the method and constructor bodies. Also, auto-rewrite lemmas are generated, which can be used conveniently during proofs (and hence reduce the proof interaction for a user).

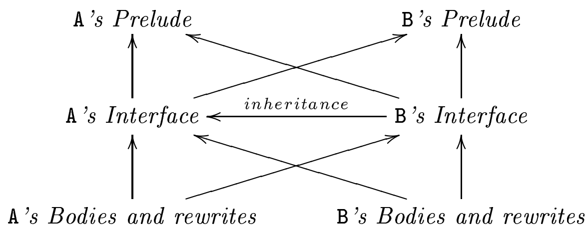


Figure4. Generated theories and their importings for the classes `A` and `B` from Figure 3

An *Interface* theory imports all *Prelude* theories of those classes which are used in its members' signatures. Moreover, it also imports the *Interface* theory of its direct superclass, since the members of superclasses should be accessible. Importings of superclasses are transitive. A *Bodies and rewrites* theory imports all *Interface* theories from those classes of which their static type occurs in method and constructor bodies. Note that there are no circularities.

3.2 Similarity between theories

The kind (and number) of theories that are produced by the LOOP tool depends on the input language. Each language has its own specific properties, *e.g.* the theories for JML describe properties of implementations, whereas the theories for Java describe concrete implementations. Though there are differences between the kind of theories generated, the three input languages have theories in common. Actually, this similarity forms the reason for having one tool for the three languages.

[3] Actually, the semantics is spread over eight theories, but due to space restrictions only the theories generated to handle mutual recursion are presented here.

For a JML class, possibly defining specification fields and methods, an extended *Interface* theory is generated, containing these extra fields and methods. And instead of a theory with semantics for bodies of methods and constructors, a theory with properties of implementations yielding from JML's specification constructs, such as behaviour specifications and invariants, is generated.

Also, when having both a Java implementation and a JML specification, another theory is generated to relate both of them, via a suitable translation of coalgebras, making the interface types match. This makes it possible to formulate the intended proof obligation, namely that the Java implementation satisfies the JML specification.

3.3 Abstract theories

The LOOP tool generates logical theories for PVS and Isabelle. Both these tools offer a higher order logic but use a different syntax. The LOOP tool first generates theories as an abstract syntax tree, which abstracts away from these differences in syntax. This tree is built from types that cover common constructs used in higher order logic, such as function abstraction and application, and quantification.

```
type expression =
  | Expression of formula
  | Application of (expression * expression)
  | Tuple of expression list
  | ...
```

```
and formula =
  | True
  | Not of formula
  | ...
```

Secondly, a theorem prover specific unparser, or pretty printer, is applied to the abstract theories in order to generate concrete theories. Writing such an unparser is fairly easy, as illustrated below, where it is done for PVS.

```
let rec pp_pvs_expression = function
  | Expression form -> pp_formula form
  | Application (func, arg) ->
      pp_pvs_expression func;
      print_string "(";
      pp_pvs_expression arg;
      print_string ")"
  | ...
and pp_pvs_formula = function
  | True -> print_string "TRUE"
  | Not form ->
      print_string "NOT ("; pp_formula form; print_string ")"
  | ...
```

3.4 Size and speed

Translating the classes in the example in Subsection 3.1 leads to 12 Kb of PVS theories and 14 Kb of Isabelle theories for class `A`, and respectively 17 Kb and 21 Kb for class `B`. The main difference in size between the PVS and Isabelle theories is caused by the fact that in Isabelle each definition, when imported from another theory, has to be qualified with its theory name. A substantial part of these generated files consists of comments, explaining what is happening.

In general, the size of the generated theories strongly depends on the number of superclasses. Every inherited method is repeated in the *Interface* theory, and its body's semantics is recalculated[4] and added to the *Bodies and rewrites* theory. Thus, inheritance leads to a substantial increase in the size of the generated theories.

The LOOP tool can easily handle a large number of classes. Running the LOOP tool on the JDK 1.0.2 API (consisting of 215 classes, forming together over 1 Mb of source-code), only takes five seconds, to parse and type check. To produce the series of logical theories takes about 50 seconds longer, mainly consisting of writing the concrete theories to file[5].

4 Use scenarios

For a successful run of the LOOP tool a Java class has to be type correct as defined by the JLS[6]. Type incorrectness of the input will lead to abrupt termination of the LOOP tool, via an error message. A successful run leads to a series of (PVS or Isabelle) type correct logical theories.

The LOOP tool requires that every Java class that is used in an implementation (and specification) occurs in the input series. This requirement is a design decision, since automatically loading of classes can lead to uncontrolled loading of too many classes. In practice it works best to cut away, for a verification, unnecessary details, *i.e.* class definitions and method definitions not used in the final program. In this way the user can restrict the size of the generated theories. It is of importance to keep this size as small as possible, to limit the time spent on loading and type checking by the theorem prover.

Once translated, the desired properties of a Java class can be verified using a theorem prover. It is up to the user how to specify these properties: either JML specifications are used (which have the advantage of automatic translation), or hand-written specifications are formulated in the language semantics (in higher order logic). The verification of these properties goes via a combination of applying (tailor-made) rewrite lemmas and definitions, and of applying Hoare logic rules [13,10].

The LOOP tool can also generate batch files. Such a batch file contains the necessary steps for a theorem prover to take for type checking the generated theories, and for rerunning proofs. Hence, batch files are useful, and reduce user interaction. They are also used for rerunning old examples after new releases (of LOOP, PVS, or Isabelle), for compatibility checks.

[4] This recalculation is necessary in order to reason about late binding in Java, which influences the behaviour of the method execution, see [9] for details.
[5] Experiments were done on a Pentium III 500 MHz, running Linux.
[6] A JML class has to be type correct following [15].

4.1 An example

The example below illustrates the ease of using JML behaviour specifications. The constructor and methods all have a `normal_behavior` specification, which informally says that if precondition holds the method is only allowed to terminate normally in a state where the postcondition holds. The LOOP tool expresses the behaviour specifications in a specialised Hoare logic for JML. Reasoning about methods goes via applying suitable Hoare rules, as described in [13].

```
class A {
  boolean b1, b2;

  /*@ normal_behavior
    @    requires : true;
    @  modifiable : b1, b2;
    @     ensures : b == (b1 & b2);
    @*/
  A (boolean b) {
    b1 = b2 = b;
  }

  /*@ normal_behavior
    @    requires : true;
    @  modifiable : b1;
    @     ensures : b1 != b2;
    @*/
  void m() { b1 = !b2; }
}
```

```
class B {

  /*@ normal_behavior
    @   requires : true;
    @    ensures : \result == false;
    @*/
  boolean n() {
    A a = new A(true);
    a.m();
    return a.b1 & a.b2;
  }
}
```

In this example, it is easy to see that the constructor and the methods all terminate normally. Note that class `B` does not declare a constructor; a default constructor is created (see [6, § 8.2]) together with a default `normal_behavior` specification (see [15, p. 48]). Thus, to prove these classes correct a user has to validate four proof obligations, of which three—of the constructors of `A` and `B`, and of method `m`—are straightforward and can be established with automatic rewriting.

Proving correctness of a method containing method calls, like method `n`, can be established in two ways: (1) reasoning with their implementations, and (2) reasoning with their specifications. In general, the latter option is better, since it reduces the time spent on proving termination of the method calls and it enables modular verification.

For method `n` in class `B` the proof obligation reads like

$$\begin{array}{l}\text{“Spec}_A\text{ holds for an \textit{arbitrary} implementation”}\\ \quad\Longrightarrow\\ \text{“Spec}_B\text{ holds for the \textit{given} implementation”.}\end{array}$$

The specification of class `A` is used to obtain the specifications of its members. These specifications are used to establish the postcondition of method `n`. Using the composition rule from [13] also this proof is straightforward.

4.2 Strategies

Currently, most of the reasoning about JML-annotated Java programs is done in PVS. As experience is growing, more and more ingredients of the verification work are being incorporated in tailor-made proof strategies in PVS. These turn out to be extremely useful and substantially decrease the amount of interaction needed for proof construction.

5 Application areas

The LOOP tool is applied in those areas, where the effort spent on specification and verification is justifiable. One can think of areas where economical and security aspects play an important role, such as the development of safety-critical systems, and integrated software development relying on formal methods.

Java's class library has many classes which are interesting for verification. Verifying classes from this class library can be useful, since many people use these classes to write their applications. The LOOP tool has been successfully applied to verify a non-trivial invariant property of the frequently used `Vector` class [11].

Also in the area of smart cards formal verification is becoming necessary, due to the higher standards the market demands. Smart cards are being issued in large numbers for security-sensitive applications, which justifies the application of formal methods: any error detected before issuing saves lots of money. The LOOP tool is used in the area of JavaCard based smart cards, especially to the JavaCard API (for specification and verification [4]), and to its applets—smart card programs—which are stored on the smart card. This work is part of a larger project, which is supported by the European Union[7].

6 Conclusions

We have presented the modular architecture of the LOOP tool, which is used to reason about Java. The LOOP tool translates the implementation and specification of Java classes into their semantics in higher order logic. Internally, this semantics is abstractly generated as a series of theories, which can easily be concretised as theories for different theorem provers. The actual verification is done in the theorem prover.

Doing full program verification for real-life programming languages is becoming feasible in more cases, but it still requires a major investment of time and resources. Such a verification technique can (only) be applied in areas where the presence of errors has a major impact on the money it costs to repair them. With a compiler like the LOOP tool, users can concentrate on the real work (specification and verification), without having to care about the actual modelling.

[7] See: `http://www.verificard.org`

Credits

Over the past couple of years many other people (than the authors) have contributed to the implementation of the LOOP tool: Ulrich Hensel, Hendrik Tews, Marieke Huisman, Martijn van Berkum, Erik Poll, Wim Janssen, Jan Rothe, and Harco Kuppens. The authors have done most of the work for the Java and JML implementation.

Acknowledgements

Thanks are due to Erik Poll and the referees for valuable comments on an earlier version.

References

1. W. Ahrendt, T. Baar, B. Beckert, M. Giese, E. Habermalz, R. Hähnle, W. Menzel, and P.H. Schmitt. The KeY approach: Integrating object oriented design and formal verification. In G. Brewka and L.M. Pereira, editors, *Proc. 8th European Workshop on Logics in AI (JELIA)*, Lect. Notes AI. Springer, October 2000.
2. B. Barras, S. Boutin, C. Cornes, J. Courant, J.-Chr. Filliâtre, E. Giménez, H. Herbelin, G. Huet, C. Muñoz, C. Murthy, C. Parent, C. Paulin-Mohring, A. Saïbi, and B. Werner. The Coq Proof Assistant User's Guide Version 6.1. Technical Report 203, INRIA Rocquencourt, France, May 1997.
3. J. van den Berg, M. Huisman, B. Jacobs, and E. Poll. A type-theoretic memory model for verification of sequential Java programs. In D. Bert and C. Choppy, editors, *Recent Trends in Algebraic Development Techniques*, number 1827 in Lect. Notes Comp. Sci., pages 1–21. Springer, Berlin, 2000.
4. J. van den Berg, B. Jacobs, and E. Poll. Formal specification and verification of JavaCard's Application Identifier Class. Techn. Rep. CSI-R0014, Comput. Sci. Inst., Univ. of Nijmegen. Appeared in: Proceedings of the JavaCard Workshop, Cannes. INRIA Techn. Rep. Updated version will appear in: I. Attali and Th. Jensen, editors, *Proceedings of the Java Card 2000 Workshop* (Springer LNCS 2001), Sept. 2000.
5. J. Corbett, M. Dwyer, J. Hatcliff, S. Laubach, C. Pasareanu, Robby, and H. Zheng. Bandera: extracting finite-state models from Java source code. In *Proceedings 22nd International Conference on Software Engineering*, June 2000.
6. J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification Second Edition.* The Java Series. Addison-Wesley, 2000.
7. 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.
8. M. Huisman. *Reasoning about JAVA Programs in higher order logic, using PVS and Isabelle*. PhD thesis, Univ. Nijmegen, 2001.
9. M. Huisman and B. Jacobs. Inheritance in higher order logic: Modeling and reasoning. In M. Aagaard and J. Harrison, editors, *Theorem Proving in Higher Order Logics*, number 1869 in Lect. Notes Comp. Sci., pages 301–319. Springer, Berlin, 2000.

10. M. Huisman and B. Jacobs. Java program verification via a Hoare logic with abrupt termination. In T. Maibaum, editor, *Fundamental Approaches to Software Engineering*, number 1783 in Lect. Notes Comp. Sci., pages 284–303. Springer, Berlin, 2000.
11. M. Huisman, B. Jacobs, and J. van den Berg. A case study in class library verification: Java's Vector class. Techn. Rep. CSI-R0007, Comput. Sci. Inst., Univ. of Nijmegen. To appear in *Software Tools for Technology Transfer*, 2001.
12. B. Jacobs. A formalisation of Java's exception mechanism. Techn. Rep. CSI-R0015, Comput. Sci. Inst., Univ. of Nijmegen. To appear at ESOP'01., 2000.
13. B. Jacobs and E. Poll. A logic for the Java Modeling Language JML. Techn. Rep. CSI-R0018, Comput. Sci. Inst., Univ. of Nijmegen. To appear at FASE'01., 2000.
14. B. Jacobs, J. van den Berg, M. Huisman, M. van Berkum, U. Hensel, and H. Tews. Reasoning about classes in Java (preliminary report). In *Object-Oriented Programming, Systems, Languages and Applications*, pages 329–340. ACM Press, 1998.
15. G.T. Leavens, A.L. Baker, and C. Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Techn. Rep. 98-06, Dep. of Comp. Sci., Iowa State Univ. (`http://www.cs.iastate.edu/~leavens/JML.html`), 1998, revised May 2000.
16. X. Leroy. *The Objective Caml system release 3.00.* Institute National de Recherche en Informatique et Automatique, 1997. Documentation and user's manual.
17. J. Meyer and A. Poetzsch-Heffter. An architecture for interactive program provers. In S. Graf and M. Schwartzbach, editors, *TACAS00, Tools ans Algorithms for the Construction and Analysis of Software*, volume 276 of *Lect. Notes Comp. Sci.*, pages 63–77, 2000.
18. D. von Oheimb and T. Nipkow. Machine-checking the Java specification: Proving type-safety. In Jim Alves-Foss, editor, *Formal Syntax and Semantics of Java*, volume 1523 of *LNCS*, pages 119–156. Springer, 1999.
19. S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In D. Kapur, editor, *11th International Conference on Automated Deduction (CADE-11)*, number 607 in Lect. Notes Comp. Sci., pages 748–752. Springer, Berlin, 1992.
20. L.C. Paulson. *Isabelle - a generic theorem prover.* Number 828 in Lect. Notes Comp. Sci. Springer, Berlin, 1994. With contributions by Tobias Nipkow.
21. LOOP Project. `http://www.cs.kun.nl/~bart/LOOP/`.
22. J. Rothe, H. Tews, and B. Jacobs. The coalgebraic class specification language CCSL. Technical Report TUD-FI00-09, Dresden University of Technology, Department of Computer Science, October 2000. Available via `http://wwwtcs.inf.tu-dresden.de/TU/Informatik/Fak/berichte.html`.
23. Extended static checker ESC/Java. Compaq System Research Center. `http://www.research.digital.com/SRC/esc/Esc.html`.