# JACK — a tool for validation of security and behaviour of Java applications*

Gilles Barthe[1], Lilian Burdy, Julien Charles[1], Benjamin Grégoire[1], Marieke Huisman[1], Jean-Louis Lanet[2], Mariela Pavlova[3]**, and Antoine Requet[2]

[1] INRIA Sophia Antipolis, France
[2] gemalto, France
[3] Ludwig-Maximilians-Universität München, Germany

**Abstract.** We describe the main features of JACK (Java Applet Correctness Kit), a tool for the validation of Java applications, annotated with JML specifications. JACK has been especially designed to improve the quality of trusted personal device applications. JACK is fully integrated with the IDE Eclipse, and provides an easily accessible user interface. In particular, it allows to inspect the generated proof obligations in a Java syntax, and to trace them back to the source code that gave rise to them. Further, JACK provides support for annotation generation, and for interactive verification. The whole platform works both for source code and for bytecode, which makes it particularly suitable for a proof carrying code scenario.

## 1 Introduction

*Motivation* Over the last years, the use of trusted personal devices (TPD), such as mobile phones, PDAs and smart cards, has become more and more widespread. As these devices are often used with security-sensitive applications, they are an ideal target for attacks. Traditionally, research has focused on avoiding hardware attacks, where the attacker has physical access to the device to observe or tamper with it. However, TPD are more and more connected to networks and moreover, provide support to execute complex programs. This has increased the risk of logical attacks, which are potentially easier to launch than physical attacks (they do not require physical access, and are easier to replicate from one device to the other), and may have a huge impact. In particular, a malicious attacker spreading over the network and massively disconnecting or disrupting devices could have significant consequences.

An effective means to avoid such attacks is to improve the quality of the software deployed on the device. This paper describes JACK[1] (the Java Applet

* This work is partially funded by the IST programme of the European Commission, under the IST-2003-507894 Inspired and IST-2005-015905 Mobius projects.

** Research done while at INRIA Sophia Antipolis.

[1] See `http://www-sop.inria.fr/everest/soft/Jack/jack.html` for download instructions.

Correctness Kit), a tool that can be used to improve the quality of applications for TPD. Such devices typically implement the Java Virtual Machine (or one of its variations)[2]. Therefore JACK is tailored to applications written in Java (bytecode). However, the described techniques are also relevant to other execution platforms for TPD.

*Characteristics of JACK* JACK allows to verify Java applications that are annotated with the Java Modeling Language (JML)[3]. An advantage of using JML is that there is wide range of tools and techniques available that use JML as specification language, *i.e.*, for testing, simulation and verification (see [11] for an overview). We distinguish two kinds of verification: at runtime, using `jmlc`, or statically. Several tools provide static verification of JML-annotated programs, adopting different compromises between soundness, completeness and automation (Section 8 provides an overview of related work). JACK implements a weakest precondition calculus, that automatically generates proof obligations that can be discharged both by automatic and interactive theorem provers. The automatic prover that is used is Simplify [22], the interactive theorem prover that is used is Coq [39].

The development of the JACK tool started in 2002 at the formal methods research laboratory of the French smart card producer Gemplus (now part of gemalto). Successful case studies with ESC/Java [17] and the LOOP tool [8] on an electronic purse smart card application [10] had sufficiently demonstrated that verification of JML annotations could help to increase the quality of smart card applications. However, the existing tools where either not precise enough, or too cumbersome to use to expose application developers to them. The JACK tool was designed to overcome these problems, in particular via the integration of JACK within the IDE Eclipse[4], and the development of a special JACK perspective.

In 2003, the tool has been transfered to the Everest project at INRIA Sophia Antipolis, and been further developed within this team since then. The other features of JACK described in this paper have been developed after this transfer.

The main characteristics of JACK that distinguish it from other static verification tools are the following:

- full integration within Eclipse IDE, including the development of a special JACK perspective that allows to inspect the different proof obligations, and from where in the code they originate;
- implementation of annotation generation algorithms: to generate "obvious" annotations, and to encode high-level security properties;
- support for verification of bytecode programs; and

[2] The standard Java set-up for TPD is the Connected Limited Device Configuration, see `http://java.sun.com/products/cldc/`, together with the MIDP profile, see `http://java.sun.com/products/midp/`.

[3] See `http://www.jmlspecs.org`.

[4] See `http://www.eclipse.org`.

– support for interactive verification, by the development of an interface and tactics for Coq and by use of the `native` construct, that allows to link JML specifications with the logic of the underlying theorem prover.

This paper illustrates how these characteristics make JACK particularly suited for the development of secure applications for TPD.

*Application Scenarios* JACK provides different kinds of support for the application developer, ranging from the automatic verification of common security properties to the interactive verification of complex functional specifications.

To support the automatic verification of high-level security properties, JACK provides an algorithm to automatically generate annotations encoding such properties, and to weave and propagate these in the application. These annotations give rise to proof obligations, whose discharge (typically automatic) guarantees adherence to the security policy. Since JACK also provides support for the verification of bytecode, and allows to compile source code level JML annotations into bytecode level specifications (written in the Bytecode Modeling Language (BML) [12]), this enables a proof carrying code scenario [35]. In such a scenario, the applications come equipped with a specification and a proof that allow the client to establish trust in the application. Since the applications usually are shipped in bytecode format, also the specification and the verification process need to be defined at this level. This scenario is even further facilitated by the fact that the compiler from JML to BML provided by JACK basically preserves the generated proof obligations (see also [6]). Thus, a software developer can verify its applications at source code level, and ship them with compiled bytecode level specifications and proofs. Notice that, provided the proof obligations can be discharged automatically, this whole process is automatic.

However, as JACK is a general-purpose tool, it can be also be used to verify complex functional-behaviour specifications. For this, it provides advanced support for specification development and interactive verification. Because of the tight integration with Eclipse, the developer does not have to change tools to validate the application. A special JACK view is provided, that allows to inspect the generated proof obligations in different views (in a Java-like syntax, or in the language of the prover). Moreover, syntax colouring of the original source code allows to see to which parts of the application and specification the proof obligation relates. Further, JACK can generate "obvious" annotations that are easy to forget, in particular preconditions that are sufficient to avoid runtime exceptions. This helps to overcome one of the major drawbacks of using JML-like annotations for specifications, namely that writing annotations is labour-intensive and error-prone. Finally, to support interactive verification, several advanced Coq tactics have been developed, and a Coq editor has been integrated into Eclipse. In addition, to be able to write expressive specifications, a `native` construct has been proposed for JML, that allows to link JML constructs directly with the logic of the underlying prover. This allows to develop the theory about these constructs directly in the logic of the theorem prover, which makes specification and verification simpler.

*Overview of the Paper* The next section gives a quick overview of the relevant JML features. Section 3 briefly outlines the general architecture of JACK, while Section 4 focuses on its user interface. Section 5 describes the different annotation generation algorithms that JACK implements. Section 6 presents the bytecode subcomponents of JACK, while Section 7 explains the features that JACK provides to support interactive verification. Finally, Section 8 concludes and discusses how this work will be continued.

Parts of the results described in this paper have been published elsewhere: [14] describes the general architecture of JACK, [37] the annotation generation algorithm for security policies, [13] the framework for the verification of bytecode, and [16] the native construct. However, this is the first time a complete overview of JACK and its main features are given in a single paper.

## 2 A Quick Overview of JML

This section gives a short overview of JML, by means of an example. Throughout the rest of this paper, we assume the reader is familiar with JML, its syntax and semantics. For a detailed overview of JML we refer to the reference manual [31]; a detailed overview of the tools that support JML can be found in [11]. Notice that JML is designed to be a general specification language that does not impose any particular design method or application domain [29].

To illustrate the different features of JML, Figure 1 shows a fragment of a specification of class `QuickSort`. It contains a public method `sort`, that sorts the array stored in the private field `tab`. Sorting is implemented via a method `swap`, swapping two elements in the array, and a private method `sort`, that actually implements the quicksort algorithm.

In order not to interfere with the Java compiler, JML specifications are written as special comments (tagged with `@`). Method specifications contain preconditions (keyword `requires`), postconditions (`ensures`) and frame conditions (`assignable`). The latter specifies which variables *may* be modified by a method. In a method body, one can annotate all statements with an `assert` predicate and loops also with invariants (`loop_invariant`), and variants (`decreases`). One can also specify class invariants, *i.e.*, properties that should hold in all visible states of the execution, and constraints, describing a relation that holds between any two pairs of consecutive visible states (where visible states are the states in which a method is called or returned from).

The predicates in the different conditions are side-effect free Java boolean expressions, extended with specification-specific keywords, such as `\result`, denoting the return value of a non-void method, `\old`, indicating that an expression should be evaluated in the pre-state of the method, and the logical quantifiers `\forall` and `\exists`. Re-using the Java syntax makes the JML specifications easily accessible to Java developers.

JML allows further to declare special specification-only variables: logical variables (with keyword `model`) and so-called `ghost` variables, that can be assigned to in special `set` annotations.

```
public class QuickSort {
  private int [] tab;

  public QuickSort(int[] tab) {this.tab = tab;}

  /*@ requires (tab != null) ;
    @ assignable tab[0 .. (tab.length -1)];
    @ ensures (\forall int i, j; 0 <= i && i <= (tab.length - 1) ==>
    @                            0 <= j && j <= (tab.length - 1) ==>
    @                            i < j ==> tab[i] <= tab[j]) &&
    @         (\forall int i; 0 <= i && i <= (tab.length - 1) ==>
    @             (\exists int j; 0 <= j && j <= (tab.length - 1) &&
    @                             \old(tab[j]) == tab[i])); @*/
  public void sort() {if(tab.length > 0) sort(0, tab.length -1);}

  /*@ requires (tab != null) && (0 <= i) && (i < tab.length) &&
    @          (0 <= j) && (j < tab.length);
    @ assignable tab[i], tab[j];
    @ ensures tab[i] == \old(tab[j]) && (tab[j] == \old(tab[i])); @*/
  public void swap(int i, int j) { ... }

  private void sort(int lo, int hi) { ... }
}
```

**Fig. 1.** Fragment of class `QuickSort` with JML annotations

Figure 1 specifies that method `sort` sorts the array `tab` from low to high, and all elements that occurred in the array initially also occur in its afterwards[5], and that method `swap` swaps the contents of the array at positions `i` and `j`.

## 3 General Architecture of JACK

This section describes the general architecture of JACK, and how it aims at a high level of precision. The next section then discusses how JACK has been made accessible to application developers by integration within the IDE Eclipse, and the development of the special JACK perspective. For the development of the JACK architecture, the main design principles were the following:

- integration within a widely-used IDE, so that developers do not have to learn a new environment, and do not have to switch between tools;
- automatic generation of proof obligations by implementation of a weakest precondition (wp) calculus;
- proof obligations are first-order logic formulae; and
- prover independence, *i.e.*, proof obligations for a single application can be verified with different provers.

[5] Note that this specification does not require that the final value of `tab` is a sorted permutation of its initial value. However, this could be expressed in JML as well.

The wp-calculus that is implemented is a so-called "direct" calculus, meaning that it works directly on an AST representation of the application, and it does not use a transformation into guarded commands, as is done by *e.g.*, ESC/Java. The wp-calculus is based on the classical wp-calculus developed by Dijkstra [23], but adapted to Java by extending it with side-effects, exceptions and other abrupt termination constructs (*cf. e.g.*, [28]). Method invocations are abstracted by their specifications, since we want verification to be modular. This direct wp-calculus has the advantage that it is easy to generate proof obligations for each path through a method, and then to connect the proof obligation with the path through the method that gave rise to this particular proof obligation (to achieve this, also some program flow information is associated to each proof obligation). This connection makes the understanding of the generated proof obligations easier. Another advantage of this approach is that the algorithms for annotation generation as described below in Section 5 could make direct use of the weakest precondition infrastructure. A drawback of this approach is that the size of the generated proof obligation may be exponential in the size of the code fragment being checked [27].

To avoid this blow up in the size of the proof obligation, and to ensure that proof obligations can be generated automatically, JACK uses several new specification constructs, introduced in [14]: loops can be annotated with frame conditions (`loop_modifies`) and exceptional postconditions (`loop_exsures`), and any code block can be specified with a block specification (similar to a method specification). The loop frame condition is used in the the wp-calculus to make a universal quantification over the loop invariant when generating the appropriate proof obligations. Block specifications and `loop_exsures` clauses improve readability and reduce the number of proof obligations, because they reduce the number of paths through a method that have to be considered.

JACK generates its proof obligations in an abstract formula language, representing first-order logic formulae. It is straightforward to translate the abstract formulae into a proof obligation for a particular prover. Adding a new prover as a plug-in to the tool is simple: one develops a background theory formalising Java's type system and memory model, and one defines how the abstract formulae are translated into concrete proof obligations for this particular prover. Initially, JACK was designed to use the AtelierB prover [1], now Simplify and Coq are the best supported back-end provers for JACK.

## 4 JACK's User Interface

One of the features that distinguish JACK from other program verification tools is the integration in the IDE Eclipse. This ensures a seamless integration of formal methods in the application development process: the application developer does not have to learn the peculiarities of a new tool, and does not have to switch tools to apply formal verification techniques.

The integration in Eclipse consists of two parts: an extension of the standard Java perspective with special JACK-related actions (checking a specification,

calling an automatic prover *etc.*), and a special JACK perspective to inspect the generated proof obligations.

### 4.1 Extension of the Java Perspective in Eclipse

The standard Java perspective of Eclipse is extended with several JACK-specific features. Menus are added to set the defaults for the different specification constructs. Further, there are buttons and menu-options to "compile" a JML specification, (*i.e.*, type check and generate proof obligations), call an automatic prover on all the generated proof obligations (either Simplify or a special Coq tactic), or change to the special JACK perspective.

Checking the JML specification is not done in a background mode, while editing the file (as is done for the type checking of Java); instead the user has to launch this action explicitly. At the time this interface was developed, adding such automatic checks required too many changes to the internals of Eclipse, which were not default available. However, in the mean time such a feature has been developed within the JMLEclipse project[6]. This project also provides syntax highlighting of JML specifications in Eclipse's Java perspective. All this could be integrated with the JACK interface.

Finally, another important constraint is the interface's responsiveness. An IDE is supposed to be used interactively, and the developer should never have to wait long for a result. Proof obligation generation is no problem for this, but calling an automatic prover on the generated proof obligations can take a significant amount of time. Therefore, the prover is called in a non-blocking way, launching a special window that allows to see the progress of the task.

### 4.2 A Proof Obligation Inspection Perspective

An important feature of JACK is that one can inspect the different generated proof obligations. Moreover, one does not have to understand the specific specification language of the prover that is being used; instead the proof obligations can be viewed in a Java/JML-like syntax (but of course, one can also choose to see the proof obligations as they are generated for a specific theorem prover).

Figure 2 shows the inspection of a proof obligation for the method `sort` in the `QuickSort` example of Figure 1. The left upper windows allows one to browse the proof obligations for the current class. Proven obligations are ticked, the others are marked with a cross. The right window shows the original source code, where the path through the code that corresponds to the current proof obligation is coloured, together with the relevant part of the method specification. Different colours are used to indicate different cases, *i.e.*, to distinguish normal from exceptional execution, and to mark that extra information, such as a method specification, or the result of a conditional expression, is available. For example, in Figure 2 one sees the specification of the private `sort` method in a pop-up box, used in the public `sort` method.

[6] See `http://jmleclipse.projects.cis.ksu.edu/`.

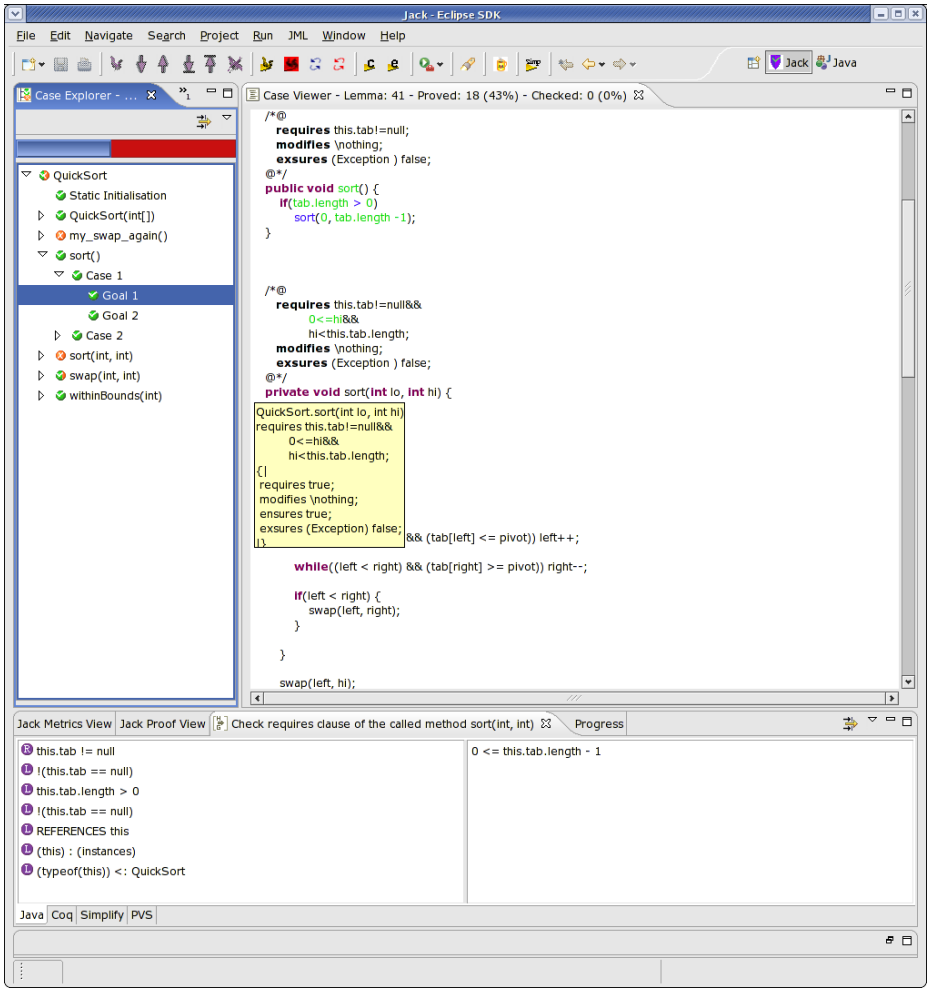


**Fig. 2.** JACK's proof obligation inspection perspective

The bottom window shows the proof obligation: the left half contains the hypotheses, marked with letters indicating their origin, *e.g.*, a hypothesis marked R originates from the method's requires clause, while a hypothesis marked L is derived from local declarations within the method. The right half of the window shows the actual goal that has to be proven. The window name highlights once again that this proof obligation originates from the postcondition. Finally, notice that the proof obligation is displayed in Java syntax, but buttons are available to change to Coq, Simplify or PVS syntax.

The user can use the proof obligation inspection view to inspect the different (unproven) proof obligations, and to launch different (interactive or specialised) provers to prove the remaining proof obligations.

```
/*@ requires this.tab!=null;
    signals (Exception) false;
  @*/
public void sort() {if(tab.length > 0) sort(0, tab.length -1);}

/*@ requires this.tab!=null && 0<=j && j<this.tab.length &&
             0<=i && i<this.tab.length;
    signals (Exception) false;
 @*/
public void swap(int i, int j) {int tmp; tmp = tab[i];
                                tab[i] = tab[j]; tab[j] = tmp;}
```

**Fig. 3.** Obvious annotations generated for a fragment of class `QuickSort`

## 5 Generating JML Annotations

While JML is easily accessible to Java developers, actually writing the specifications of an application is labour-intensive and error-prone, as it is easy to forget some annotations. There exist tools which assist in writing these annotations, *e.g.*, Daikon [25] and Houdini [26] use heuristic methods to produce annotations for simple safety and functional invariants. However, these tools cannot be guided by the user—they do not require any user input—and in particular cannot be used to generate annotations from realistic security policies.

Within JACK, we have implemented several algorithms to generate annotations. We can distinguish two goals for annotation generation. The first is to reduce the burden of annotation writing by generating as much "obvious" annotations as possible. Given an existing, unannotated, application, one first generates these obvious annotations automatically, before developing the more interesting parts of the specification. The second goal is to encode high-level properties by encoding these with simple JML annotations, that are inserted at all appropriate points in the application, so that they can be checked statically. JACK implements algorithms for both goals, as described in this section.

### 5.1 Generation of Preconditions

JACK implements an algorithm to generate "obvious" minimal preconditions to avoid null-pointer and array-out-of-bounds exceptions. This algorithm re-uses the implementation of the wp-calculus: it computes the weakest precondition for the specification `signals (NullPointerException) false;` (resp. `signals (ArrayIndexOutOfBoundsException) false;`) and inserts this as annotations in the code.

As an example, Figure 3 shows the annotations that are generated for some methods of the class `QuickSort` of Figure 1. It is important to realise that the specifications that are generated might not be very spectacular, but that they are generated *automatically*.

The annotation generation could be further improved by applying a simple analysis on the generated annotations. Often it is the case that the preconditions that are generated for the fields of the class are the same for (almost) all methods. In that case, this condition is likely to be a class invariant, and instead of generating a precondition for each method, it would be more appropriate to generate a single class invariant. For example, for the class `QuickSort`, this would produce an annotation `invariant this.tab!=null;`.

### 5.2 Encoding of Security Policies

Another difficulty when writing annotations is that a conceptually simple high-level property can give rise to many different annotations, scattered through the code, to encode this property. This is typically the case for many security policies. Current software practice for the development of applications for trusted personal devices is that security policies give rise to a set of *security rules* that should be obeyed by the implementation. Obedience to these rules is established by manual code inspection; however it is desirable to have tool support for this, because a typical security property may involve several methods from different classes. Many of the security rules can be formalised as simple automata, which are amenable to formal verification. Therefore, we propose a method that given a security rule, automatically annotates an application, in such a way that if the application respects the annotations then it also respects the security policy. Thus, it is not necessary for the user to understand the generated annotations, he just has to understand the security rules.

The generation of annotations proceeds in two phases: first we generate core-annotations that specify the behaviour of the methods directly involved, and next we propagate these annotations to all methods directly or indirectly invoking the methods that form the core of the security policy. The second phase is necessary because we are interested in static verification. The annotations that we generate all use only JML static ghost variables; therefore the properties are independent of the particular class instances available.

As a typical example of the kind of security rules our approach can handle, we consider the atomicity mechanism in Java Card (Java for smart cards) ([37] gives more examples of such security rules). A smart card does not include a power supply, thus a brutal retrieval from the terminal could interrupt a computation and bring the system in an incoherent state. To avoid this, the Java Card specification prescribes the use of a transaction mechanism to control synchronised updates of sensitive data. A statement block surrounded by the methods `beginTransaction()` and `commitTransaction()` can be considered atomic. If something happens while executing the transaction (or if `abortTransaction()` is executed), the card will roll back its internal state to the state before the transaction was begun. To ensure the proper functioning and prevent abuse of this mechanism, applications should respect for example the following security rules.

**No nested transactions** Only one level of transactions is allowed.

**No exception in transaction** All exceptions that may be thrown inside a transaction, should also be caught inside the transaction.

**Bounded retries** No pin verification may happen within a transaction.

The second rule ensures that a transaction will always be closed; if the exception would not be caught, `commitTransaction` would not be executed. The last rule avoids the possibility to abort the transaction every time a wrong pin code has been entered. As this would roll back the internal state to the state before the transaction was started, this would also reset the retry counter, thus allowing an unbounded number of retries. Even though the specification of the Java Card API prescribes that the retry counter for pin verification cannot be rolled back, in general one has to check this kind of properties.

Such properties can be easily encoded with automata, describing in which states a certain method is allowed to be called. Based on this automata, we then generate core-annotations. For example, the atomicity properties above give rise to core-annotations for the methods related to the transaction mechanism declared in class `JCSystem` of the Java Card API. A static ghost variable

```
/*@ static ghost int TRANS == 0; @*/
```

is declared, that is used to keep track of whether there is a transaction in progress. To specify the **No nested transactions** property, the core-annotations for method `beginTransaction` are the following.

```
/*@ requires TRANS == 0;
  @ assignable TRANS;
  @ ensures TRANS == 1; @*/
public static native void beginTransaction()
                          throws TransactionException;
```

Similar annotations are generated for `commitTransaction` and `abortTransaction` ([37] also describes the generated core-annotations for the other properties). After propagation, these annotations are sufficient to check for the absence of nested transactions. To understand why propagation is necessary, suppose we are checking the **No nested transactions** property for an application, containing the following fragment (where `m` does not call any other methods, and does not contain any set-annotations).

```
void m() { ... // some internal computations
           JCSystem.beginTransaction();
           ... // computations within transaction
           JCSystem.commitTransaction(); }
```

When applying static verification on this code fragment, the core-annotations for `beginTransaction` will give rise to a proof obligation that the precondition of method `m` implies that there is no transaction in progress, *i.e.*, `TRANS == 0` (since `TRANS` is not modified by the code that precedes the call to `beginTransaction`).

The only way this proof obligation can be established is if the precondition of `beginTransaction` is propagated as a precondition for method `m`. In contrast, the precondition for `commitTransaction` (`TRANS == 1`) does not have to be propagated to the specification of `m`; instead it has to be established by the postcondition of `beginTransaction`, because the variable `TRANS` is modified by this method.

In a similar way, the postcondition for the method `commitTransaction` is propagated to the postcondition of method `m`. This information can then be used for the verification of yet another method, that contains a call to method `m`.

The propagation method not only propagates preconditions and normal and exceptional postconditions, it also propagates assignable clauses. We have shown that the algorithm that we use corresponds to an abstract version of the wp-calculus (where we only consider static variables). We have exploited this correspondence in the implementation, by re-using the wp-calculus infrastructure to implement the propagation algorithm. For a more formal treatment of the propagation algorithm, and the correspondence statement, we refer to [37].

To illustrate the effectiveness of our approach, we tested our method on several industrial smart card applications, including the so-called Demoney case study, developed as a research prototype by Trusted Logic[7], and the PACAP case study [9], developed by Gemplus. Both examples have been explicitly developed as test cases for different formal techniques, illustrating the different issues involved when writing smart card applications. We used the core-annotations as presented above, and propagated these throughout the applications. For both applications we found that they contained no nested transactions, and that they did not contain attempts to verify pin codes within transactions. However, in the PACAP application we found transactions containing uncaught exceptions. All proof obligations generated *w.r.t.* these properties are trivial and can be discharged immediately. However, to emphasise the usefulness of having a tool for generating annotations: we encountered cases where a single transaction gave rise to twenty-three annotations in five different classes. When writing these annotations manually, it is all too easy to forget some.

## 6 Specification and Verification of Bytecode

JACK allows one to verify applications not only at source code level, but also at bytecode level. This is in particular important to support proof carrying code [35], where bytecode applications are shipped together with their specification and a correctness proof. However, the possibility to verify bytecode also has an interest on its own: sometimes security-critical applications are developed directly at bytecode level, in order not to rely on the correctness of the compiler. To be able to formally establish the correctness of such an application, one needs support to verify bytecode directly.

This section describes the different parts in the bytecode subcomponent of JACK. First, we present a specification language tailored to bytecode, and we

[7] See `http://www.trusted-logic.com`.

*predicate* ::= ...

*unary-expr-not-plus-minus* ::= ...
    | *primary-expr* [*primary-suffix*]...
*primary-suffix* ::= `.` *ident* | `(` [*expression-list*] `)` | `[` *expression* `]`
*primary-expr* ::= `#`*natural* % reference in the constant pool
    | `lv[`*natural*`]` % local variable
    | *bml-primary*
    | *constant* | `super` | `true` | `false` | `this` | `null` | `(`*expression*`)` | *jml-primary*
*bml-primary* ::= `cntr` % counter of the operand stack
    | `st(`*additive-expr*`)` % stack expressions
    | `length(`*expression*`)` % array length

**Fig. 4.** Fragment of grammar for BML predicates and specification expressions

specify how these specifications can be encoded in the class file format. Our specification language, called BML for Bytecode Modeling Language [12], is the bytecode cousin of JML. Second, we define and implement a compiler from JML to BML specifications. Such a compiler is in particular useful in a proof carrying code scenario, where the application developer can verify the application at (the more intuitive) source code level, and then compile both the application and the specification to bytecode level. Finally, we also define a verification condition generator for bytecode applications annotated with BML, implementing a wp-calculus for bytecode. This allows to generate the proof obligations for a bytecode application to satisfy its BML specification.

### 6.1 A Specification Language for Bytecode: BML

BML has basically the same syntax as JML with two exceptions:

1. specifications are not written directly in the program code, they are added as special attributes to the bytecode; and
2. the grammar for expressions only allows bytecode expressions.

Figure 4 displays the most interesting part of the grammar for BML predicates, defining the syntax for primary expressions and primary suffixes[8]. Primary expressions, followed by zero or more primary suffixes, are the most basic form of expressions, formed by identifiers, bracketed expressions *etc.*

Since only bytecode expressions can be used, all field names, class names *etc.*, are replaced by references to the constant pool (a number, preceded by the symbol `#`), while registers are used to refer to local variables and parameters. The grammar also contains several bytecode specific keywords, such as `cntr`, denoting the stack counter, `st(`$e$`)` where $e$ is an arithmetic expression, denoting the $e^{th}$ element on the stack, and `length(`$a$`)`, denoting the length of array $a$. In addition, the specification-specific JML keywords are also available.

To show a typical BML specification, Figure 5 presents the BML version of the JML specification of method `swap` in Figure 1. Notice that the field `tab` has

[8] See `http://www-sop.inria.fr/everest/BML` for the full grammar.

```
requires this.#14 != null && 0 <= lv[1] && lv[1] < length(this.#14) &&
         0 <= lv[2] && lv[2] < length(this.#14) && true
assignable this.#14.[lv[2]],this.#14[lv[1]]
ensures this.#14[lv[1]] == \old(this).\old(#14)[\old(lv[2])] &&
        this.#14[lv[2]] == \old(this).\old(#14)[\old(lv[1])]
 0 aload_0
 1 getfield #14
 4 iload_1
 5 iaload
 6 istore_3
 7 aload_0
 8 getfield #14
11 iload_1
12 aload_0
13 getfield #14
16 iload_2
17 iaload
18 iastore
19 aload_0
20 getfield #14
23 iload_2
24 iload_3
25 iastore
26 return
```

**Fig. 5.** Bytecode + BML specification for method `swap` in class `QuickSort`

been assigned the number 14 in the constant pool, and that it is always explicitly qualified with `this` in the specification. In the bytecode the variable `this` is stored in `lv[0]` (thus it can be accessed by `aload_0`). The method's parameters `i` and `j` are denoted by the expressions `lv[1]` and `lv[2]`, respectively. Notice further that the BML specification directly corresponds to the original JML specification.

### 6.2 Encoding BML Specifications in the Class File Format

To store BML specifications together with the bytecode it specifies, we encode them in the class file format. The Java Virtual Machine Specification [32] prescribes the mandatory elements of the class file: the constant pool, the field information and the method information. User-specific information can be added to the class file as special user-specific attributes ([32, §4.7.1]). We store BML specifications in such user-specific attributes, in a compiler-independent format. The use of special attributes ensures that the presence of BML annotations does not have an impact on the application's performance, *i.e.*, it will not slow down loading or normal execution of the application.

For each class, we add the following information to the class file:

```
Ghost_Field_attribute {
    u2 attribute_name_index;
    u4 attribute_length;
    u2 fields_count;
    { u2 access_flags;
      u2 name_index;
      u2 descriptor_index;
    } fields[fields_count]; }
```

**Fig. 6.** Format of attribute for ghost field declarations

- a second constant pool which contains constant references for the BML specification expressions;
- an attribute with the ghost fields used in the specification;
- an attribute with the model fields used in the specification;
- an attribute with the class invariants (both static and object); and
- an attribute with the constraints (both static and object).

Apart from the second constant pool, all extra class attributes basically consist of the name of the attribute, the number of elements it contains, and a list with the actual elements. As an example, Figure 6 presents the format of the ghost field attribute. This should be understood as follows: the name of the attribute is given as an index into the (second) constant pool. This constant pool entry will be representing a string `"Ghost_Field"`. Next we have the length of the attribute and the number of fields stored in the attribute. The **fields** table stores all ghost fields. For each field we store its access flag (*e.g.*, `public` or `private`), and its name and descriptor index, both referring to the constant pool. The first must be a string, representing the (unqualified) name of the variable, the latter is a field descriptor, containing *e.g.*, type information. The tags **u2** and **u4** specifies the size of the attribute, 2 and 4 bytes, respectively. The format of the other attributes is specified in a similar way (see [36] for more details).

### 6.3 Compiling JML Specifications into BML Specifications

We have implemented a compiler from JML specifications into BML specifications – stored in the class file. The JML specification is compiled separately from the Java source code. In fact, the compiler takes as input an annotated Java source file *and* the class file produced by a non-optimising compiler with the debug flag set.

From the debug information, we use in particular the **Line_ Number_Table** and the **Local_Variable_Table** attributes. The **Line_Number_Table** links line numbers in the source code with the bytecode instructions, while the **Local_Variable_Table** describes the local variables that appear in a method.

JML specifications are compiled into BML specifications in several steps:

1. compilation of ghost and model field declarations;

2. linking and resolving of source data structures to bytecode structures;
3. locating instructions for annotation statements; this information is added as a special **index** entry in the attribute (a heuristic algorithm is used to find the entry point of a loop, for more details see [36]);
4. compilation of JML predicates, taking into account that not all source code level primitive types are directly supported at bytecode level; and
5. generation of user-specific class attributes.

### 6.4 Verification of Bytecode

To generate proof obligations, we have implemented a wp-calculus for bytecode in JACK. Just as the source code level wp-calculus, it works directly on the bytecode; the program is not transformed into a guarded command format. Again, this has the advantage that we can easily trace proof obligations back to the relevant bytecode and BML fragment.

The JACK implementation supports all Java bytecode sequential instructions, except for floating point arithmetic instructions and 64 bit data (`long` and `double`). Thus in particular, it handles exceptions, object creation, references and subroutines. The calculus is defined over the method's control flow graph.

The verification condition generator proceeds as follows. For each method proof obligations are generated for each execution path by applying the weakest predicate transformer to every instruction where the method might end (*i.e.*, `return` or `athrow` instructions), and at each loop exit point. The wp-calculus then follows control flow backwards, until it reaches the entry point instruction.

The weakest precondition transformer takes three arguments: the instruction for which we calculate the precondition, the instruction's normal postcondition $\psi$ and the instruction's exceptional postcondition $\phi^{Exc}$. For the full wp-calculus for BML-annotated bytecode, and its soundness proof we refer to [36]. Here we show as an example the wp-rule for the instruction $\texttt{load}_i$.

$$\texttt{wp}(\texttt{load}_i, \psi, \psi^{Exc}) = \psi[\texttt{cntr} \leftarrow \texttt{cntr} + 1][\texttt{st(cntr +1)} \leftarrow \texttt{lv[i]}]$$

Since the $\texttt{load}_i$ instruction will always terminate normally, only the normal postcondition is involved, after updating it to reflect the changes that are made to the stack, *i.e.*, the value that was stored in the local variable register `lv[i]` is now at the top of the stack (at position `st(cntr + 1)`), and the stack counter is increased.

Finally we would like to remark that there is a close correspondence between the proof obligations generated by JACK at source code level, and the proof obligations that are generated once the application and the specification are compiled at bytecode level (provided that the application is compiled with a non-optimising compiler): modulo names and the handling of shorts, bytes and boolean values, the proof obligations are equivalent. This means that proofs for proof obligations at source code level can be re-used for proof obligations at bytecode level (see also [6] for a compilation of source code level proofs to

bytecode level proofs). This is in particular important for the proof carrying code scenario [35], where the code producer develops a proof at source code level, and then ships bytecode level application and specification. Modulo the necessary re-namings, the proofs can be shipped directly, and the code client can verify these using a verification condition generator at bytecode level.

## 7 Support for Interactive Verification

When verifying complex functional behaviour specifications, automatic provers often fail to solve the proof obligations. In that case, the user can instead try to solve the proof obligation interactively (or, in case the proof obligations is unprovable, analyse it thoroughly to find the source of the error). JACK provides support for interactive verification using the Coq proof assistant [19].

This section first discusses the special features of JACK's Coq plug-in to support interactive verification, and the special Coq editor integrated in Eclipse, then it presents JACK's specific annotation keyword for interactive verification: the `native` keyword.

### 7.1 The Coq Plug-In

Proof readability and proof re-usability is crucial in interactive verification, in contrast to automatic verification where proof obligations are simply sent to the automatic prover and it is of no importance whether the proof obligation is human-readable or not. Therefore we developed a set of facilities for pretty printing proof obligations, to reuse the proofs – in particular to allow replaying the proofs when the specifications have changed – and for proof construction.

JACK uses short variable names in proof obligations as much as possible, but in case of ambiguity long variable names are used. Basically, JACK generates all variable names for all proof obligations of one file in one go. However, for interactive verification the variables only have to be considered within the scope of a single proof obligation, thus short variable names can be used more often. Therefore, the Coq plug-in re-disambiguates per proof obligation. This results in better proof readability as the variable names are shorter. The main pretty printing is done directly through Coq's own pretty printing features.

Special attention has also been given when storing the proof obligations to a file. First, the file has a human-readable name, so the user can easily retrieve the proof obligation as well as its proof script. If the lemma is regenerated or reopened, he can step through the proof (and adapt it if necessary), and it does not have to be rewritten from scratch. The different kinds of hypotheses are separated from each other and are given different names. This is in particular important when a proof obligation has been modified (and is considered unproved) by a change to the specification: if the proof did not involve any of the modified hypotheses, it remains valid. This facilitates greatly the reuse of proofs.

One of the key points in interactive verification is the level of difficulty to manipulate the proof assistant in order to construct a proof. To help the user

build proof scripts that are both intuitive to read and to make, we have used the tactic mechanism of Coq [21]. As JACK originally generates proof obligations for automatic verification, numerous hypotheses are added to help the automatic theorem prover. For interactive verification these hypotheses are often useless (and annoying). Therefore we have developed tactics to clean up the proof obligations. There are also tactics[9] to solve common proof patterns generated by JACK: (*i*) to solve arithmetic goals, (*ii*) to solve proofs by contradiction, (*iii*) to solve array-specific proof obligations, and (*iv*) to solve proof obligations related to assignments. Finally, the Coq plug-in also allows automatic resolution of proof obligations using generic proof scripts, and application-specific tactics can be defined to be used both for interactive verification and with automatic resolution.

### 7.2 JACK with Coq in Eclipse

An important feature of JACK is that all development can be done inside Eclipse. Therefore, the Coq plug-in contains an editor for Coq, called CoqEditor. CoqEditor provides a way to interact directly with Coq through Eclipse's Java environment, so the user can process and edit Coq files (containing proof scripts or user-defined tactics). CoqEditor resembles the Isabelle plug-in in Proof General Eclipse [40], but uses a more light-weight approach. It has keyboard shortcuts similar to CoqIde (the current Coq graphical interface, written in OCaml[10]). Of course, it provides syntax highlighting and one can interactively process a Coq file. In addition, CoqEditor has an outline view, that summarises the structure of the currently edited Coq file in a tree-like representation (this is especially useful to see the modules hierarchy), and an incremental indexing feature, that allows the user to jump directly from a keyword to its definition.

### 7.3 Native specifications

When specifying complex applications, often one needs advanced data structures. It is a major challenge how to specify these in a way that is suitable for verification (see Challenge 1 in [30]). A possible way to do this in JML is by using so-called model classes, but this makes verification awkward, because all operations on these data structures have to be specified by pre-post-condition specifications. A more convenient approach is to use constructs that are specific to the logic of the prover in which the proofs will be developed. This is exactly the functionality provided by the `native` construct [16], *i.e.*, it relates declarations in the JML specification directly to the logic of the underlying prover. We have implemented the `native` construct for Coq, but the same principle can be used to support any other prover.

[9] See `http://www-sop.inria.fr/everest/soft/Jack/doc/plug-in/coq/Prelude/` for a full description of the different tactics.

[10] Available via the Coq distribution (`http://coq.inria.fr`).

In JML we define:

```
/*@ public native class IntList {
     public native IntList append (IntList l);
     public native static IntList create();
     ...
     public native static IntList toList (int [] tab);
   } @*/
```

And in Coq:

```
Definition IntList := list t_int.
Definition IntList_create: IntList := nil.
Definition IntList_append: list t_int -> list t_int -> list t_int := app.
...
```

**Fig. 7.** The definition of the native type `IntList`

```
//@ ghost IntList list;

/*@ requires (tab!=null) && list.equals(IntList.toList(tab))
    assignable tab[0 .. (tab.length -1)], list;
    ensures list.equals(IntList.toList(tab)) &&
              list.isSorted() && list.isInjection(\old(list));
 @*/
public void sort() {if(tab.length > 0) sort(0, tab.length -1);}
```

**Fig. 8.** Specification of method `sort` with `native` construct

The `native` construct can be used for types and methods. A `native` method is a specification-only method that has no body and no (JML) specification. It must terminate normally and cannot have any side-effects. A `native` type is a type to use with specification methods (`native` methods as well as JML's `model` methods). Both are related to constructs defined in the proof obligation's target language: `native` types are bound to types and `native` methods to function definitions. When generating the proof obligations, the `native` constructs are treated as uninterpreted function symbols. The user then specifies in a Coq file how the function symbols are bound to constructs in Coq.

For example, Figure 7 defines a `native` type IntList and binds it to the type of list of integers in Coq. This allows to use Coq's list library in the proofs. Using the `native` type declaration, the specification of the `sort` method (from Figure 1 on page 5) can be rewritten as in Figure 8. Notice that this results in more readable and natural annotations, because instead of relying on arrays, we can write it directly in the proof obligation's target language syntax. The use of the `native` construct also allows the user to define more easily auxiliary lemmas that can be used to prove the proof obligations and to add automation to proof scripts.

## 8 Conclusions

This paper describes the main characteristics of JACK, the Java Applet Correctness Kit, a tool set for the validation of security and functional behaviour properties for Java applications. We have focused in particular on the features that distinguish JACK from other similar tools:

- the integration into a standard IDE;
- a user interface that helps to understand the proof obligations;
- the implementation of an algorithm to generate "obvious" annotations;
- the implementation of an algorithm to encode high-level security properties with JML annotations;
- support for the verification of both source code *and* bytecode; and
- support for interactive verification, both practical (development of user interface and tactics) and theoretical (`native` construct to link annotations with the logic of the underlying theorem prover).

The JACK tool has been used for several small to medium-scale case studies. First of all, we have shown how BML annotations can be used to guarantee resource policies related to memory consumption of bytecode applications [5]. In addition, we have also shown how the verification of exception-freeness at bytecode level can be used to reduce the footprint of Java-to-native compilation schemes. Executable code typically contains run-time checks to decide whether an exception should be thrown. But if it can be proven statically that the exception *never* will be thrown, there is no need for the executable code to contain the run-time checks [20].

Development and maintenance of a verification tool for a realistic programming language is a major effort. During the last decade several such tools have been developed (see the related work section below). This has resulted in a drastic improvement of the technologies available to verify applications. However, we believe that now the moment has come to combine the different technologies, and to bundle this into one powerful verification tool. Development of such a tool is one of the goals of the IST Mobius project. It is foreseen that all technology developed around JACK that distinguish it from other verification tools will be integrated in this single verification tool.

*Related work* Several other tools exist aiming at the static verification of JML-annotated Java code, but JACK distinguishes itself from these tools by the features described in this paper. We briefly describe the most relevant other tools. ESC/Java [17] is probably the most used tool. It also aims at a high level of automation, but makes an explicit trade-off between soundness, completeness and automation. The Jive tool [34] uses a Hoare-logic for program verification, requiring much more user interaction. The Key tool [7] uses a dynamic logic approach, where program verification resembles program simulation. The LOOP tool [8] translates both the program and the annotations into specifications in the logic of the PVS theorem prover. Both a Hoare logic and a weakest precondition

calculus have been proven sound in PVS, and can be used to verify whether the program respects its specification.

Krakatoa [33] translates JML-annotated programs into an intermediate format, for which the Why tool generates proof obligations. Krakatoa allows the user to specify algebraic specifications as part of the annotations, and use these in the verification. This resembles the `native` construct, however, the `native` construct directly allows one to use the full expressiveness of the logic of the underlying prover, and to directly reuse any (library) results already proven about the data types. Moreover, the use of the native construct allows one to keep on using a Java-like syntax in the annotations.

The Spec#/Boogie project [3] aims at the specification and verification of annotated C# programs. As JACK, the tool also provides support for verification at source code and bytecode level. However, they do not compile source code specifications into bytecode specifications, that can be shipped with the application. Further, the project mainly aims at automatic verification, and does not provide support for annotation generation.

There are several projects that aim at annotating bytecode: JVer is a tool to verify annotated bytecode [15], but they do not have a special bytecode specification language. The Extended Virtual Platform project aims at developing a framework that allows to compile JML annotations, to allow run-time checking [2], but they do not allow to write specifications directly at bytecode level.

Most approaches to ensuring high-level security properties are based on run-time monitoring, see *e.g.*, [4, 38, 24, 18]. However, run-time monitoring is not an option for trusted personal devices: for the user it would be unacceptable to be blocked in the middle of an application, because of a security violation.

## References


1. J.-R. Abrial. *The B Book, Assigning Programs to Meanings*. Cambridge University Press, 1996.
2. S. Alagić and M. Royer. Next generation of virtual platforms. Article in `odbms.org`, October 2005. `http://odbms.org/about_contributors_alagic.html`.
3. M. Barnett, K.R.M. Leino, and W. Schulte. The Spec# programming system: An overview. In G. Barthe, L. Burdy, M. Huisman, J.-L. Lanet, and T. Muntean, editors, *Proceedings, Construction and Analysis of Safe, Secure and Interoperable Smart devices (CASSIS'04) Workshop*, volume 3362 of *Lecture Notes in Computer Science*, pages 151–171. Springer-Verlag, 2005.
4. D. Bartetzko, C. Fischer, M. Möller, and H. Wehrheim. Jass – Java with Assertions. In K. Havelund and G. Roşu, editors, *ENTCS*, volume 55(2). Elsevier Publishing, 2001.
5. G. Barthe, M. Pavlova, and G. Schneider. Precise analysis of memory consumption using program logics. In *Software Engineering and Formal Methods*, pages 86–95. IEEE Press, 2005.
6. G. Barthe, T. Rezk, and A. Saabas. Proof obligations preserving compilation. In R. Gorrieri, F. Martinelli, P. Ryan, and S. Schneider, editors, *Proceedings of FAST'05*, volume 3866 of *LNCS*, pages 112–126. Springer, 2005.

7. B. Beckert, R. Hähnle, and P.H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*. Number 4334 in LNCS. Springer, 2007.
8. J. van den Berg and B. Jacobs. The LOOP compiler for Java and JML. In T. Margaria and W. Yi, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2001)*, number 2031 in LNCS, pages 299–312. Springer, 2001.
9. P. Bieber, J. Cazin, P. Girard, J.-L. Lanet, V. Wiels, and G. Zanon. Checking secure interactions of smart card applets. *Journal of Computer Security*, 10(4):369–398, 2002.
10. C. Breunesse, N. Cataño, M. Huisman, and B. Jacobs. Formal methods for smart cards: an experience report. *Science of Computer Programming*, 55(1-3):53–80, 2005.
11. L. Burdy, Y. Cheon, D. Cok, M. Ernst, J.R. Kiniry, G.T. Leavens, K.R.M. Leino, and E. Poll. An overview of JML tools and applications. In T. Arts and W. Fokkink, editors, *Workshop on Formal Methods for Industrial Critical Systems*, volume 80 of *Electronic Notes in Theoretical Computer Science*, pages 73–89. Elsevier Science, Inc., 2003. Preprint University of Nijmegen (TR NIII-R0309).
12. L. Burdy, M. Huisman, and M. Pavlova. Preliminary design of BML: A behavioral interface specification language for Java bytecode. In *Fundamental Approaches to Software Engineering (FASE 2007)*, volume 4422 of *Lecture Notes in Computer Science*, pages 215–229. Springer-Verlag, 2007.
13. L. Burdy and M. Pavlova. Java bytecode specification and verification. In *Symposium on Applied Computing*, pages 1835–1839. Association of Computing Machinery Press, 2006.
14. L. Burdy, A. Requet, and J.-L. Lanet. Java applet correctness: A developer-oriented approach. In D. Mandrioli K. Araki, S. Gnesi, editor, *Formal Methods Europe*, volume 2805 of *Lecture Notes in Computer Science*, pages 422–439. Springer-Verlag, 2003.
15. A. Chander, D. Espinosa, N. Islam, P. Lee, and G. Necula. JVer: A Java Verifier. In *Conference on Computer Aided Verification (CAV'05)*, 2005.
16. J. Charles. Adding native specifications to JML. In *Workshop on Formal Techniques for Java Programs*, 2006.
17. D. Cok and J.R. Kiniry. ESC/Java2: Uniting ESC/Java and JML. In G. Barthe, L. Burdy, M. Huisman, J.-L. Lanet, and T. Muntean, editors, *Proceedings, Construction and Analysis of Safe, Secure and Interoperable Smart devices (CASSIS'04) Workshop*, volume 3362 of *Lecture Notes in Computer Science*, pages 108–128. Springer-Verlag, 2005.
18. T. Colcombet and P. Fradet. Enforcing trace properties by program transformation. In *Principles of Programming Languages, POPL'00*, pages 54–66. ACM Press, 2000.
19. Coq development team. The Coq proof assistant reference manual V8.0. Technical Report 255, INRIA, France, mars 2004. `http://coq.inria.fr/doc/main.html`.
20. A. Courbot, M. Pavlova, G. Grimaud, and J.J. Vandewalle. A low-footprint Java-to-native compilation scheme using formal methods. In J. Domingo-Ferrer, J. Posegga, and D. Schreckling, editors, *proceedings of CARDIS*, volume 3928 of *Lecture Notes in Computer Science*, pages 329–344. Springer, 2006.
21. D. Delahaye. A tactic language for the system Coq. In *LPAR*, pages 85–95, 2000.
22. D. Detlefs, G. Nelson, and J.B. Saxe. Simplify: a theorem prover for program checking. *Journal of the Association of Computing Machinery*, 52(3):365–473, 2005.
23. E.W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, 1975.

24. U. Erlingsson. *The Inlined Reference Monitor Approach to Security Policy Enforcement*. PhD thesis, Department of Computer Science, Cornell University, 2003. Available as Technical Report 2003-1916.
25. M.D. Ernst, J. Cockrell, W.G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2):1–25, 2001.
26. C. Flanagan and K.R.M. Leino. Houdini, an annotation assistant for ESC/Java. In J.N. Oliveira and P. Zave, editors, *Formal Methods Europe 2001 (FME'01): Formal Methods for Increasing Software Productivity*, number 2021 in LNCS, pages 500–517. Springer, 2001.
27. C. Flanagan and J.B. Saxe. Avoiding exponential explosion: Generating compact verification conditions. In *Principles of Programming Languages*, pages 193–205, New York, NY, USA, 2001. Association of Computing Machinery Press.
28. B. Jacobs. Weakest precondition reasoning for Java programs with JML annotations. *Journal of Logic and Algebraic Programming*, 58:61–88, 2004.
29. G.T. Leavens, A.L. Baker, and C. Ruby. Preliminary design of JML: A behavioral interface specification language for Java. *ACM SIGSOFT Software Engineering Notes*, 31:1–38, 2006.
30. G.T. Leavens, K.R.M. Leino, and P. Müller. Specification and verification challenges for sequential object-oriented programs. *Formal Aspects of Computing*, 2007. To appear.
31. G.T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. Cok, and J. Kiniry. *JML Reference Manual*, July 2005. In Progress. Department of Computer Science, Iowa State University. Available from `http://www.jmlspecs.org`.
32. T. Lindholm and F. Yellin. *The Java™ Virtual Machine Specification. Second Edition*. Sun Microsystems, Inc., 1999. http://java.sun.com/docs/books/vmspec/.
33. C. Marché, C. Paulin-Mohring, and X. Urbain. The Krakatoa tool for certification of Java/JavaCard programs annotated with JML annotations. *Journal of Logic and Algebraic Programming*, 58:89–106, 2004.
34. J. Meyer and A. Poetzsch-Heffter. An architecture of interactive program provers. In S. Graf and M. Schwartzbach, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2000)*, number 1785 in LNCS, pages 63–77. Springer, 2000.
35. G.C. Necula. Proof-carrying code. In *Principles of Programming Languages*, pages 106–119, New York, NY, USA, 1997. Association of Computing Machinery Press.
36. M. Pavlova. *Specification and verification of Java bytecode*. PhD thesis, Université de Nice Sophia-Antipolis, 2007.
37. M. Pavlova, G. Barthe, L. Burdy, M. Huisman, and J.-L. Lanet. Enforcing high-level security properties for applets. In P. Paradinas and J.-J. Quisquater, editors, *CARDIS 2004*. Kluwer Academic Publishing, 2004.
38. F.B. Schneider. Enforceable security policies. Technical Report TR99-1759, Cornell University, October 1999.
39. The Coq Development Team. *The Coq Proof Assistant Reference Manual – Version V8.1*, July 2006. `http://coq.inria.fr`.
40. D. Winterstein, D. Aspinall, and C. Lüth. Proof General/Eclipse: A generic interface for interactive proof. In *International Workshop on User Interfaces for Theorem Provers 2005 (UITP'05)*, 2005.