Methodology for Java Application Validation

INRIA Sophia Antipolis
Everest team

Contents

List of Figures

Chapter 1

Introduction

Providing high quality on applet development is becoming a crucial issue, especially when those applets are aimed to be loaded and executed in secure device like smart cards. Actually, the card remains a specific domain where post issuance corrections are very expensive due to the deployment process and the mass production. Currently, the quality is ensured by costly test campaigns, whenever tests are technically possible. We consider that using formal techniques is a solution that allows to increase the quality, but also to reduce validation costs.

Nevertheless, proving program correctness, and more generally using formal methods, is traditionally an activity reserved for experts. This restriction is usually caused by the mathematical nature of the concepts involved. This explains why formal techniques are difficult to introduce in industrial processes, even if they are now widely used in research and teaching activities. However, we believe that this restriction can be reduced by providing notations and tools hiding the mathematical formalisms. Therefore, formal tools should be developed to fit into classical developers environment. We strongly believe that efforts should be done to allow users to benefit from formal techniques without having to learn new formalisms and to become experts.

All the tools and results presented in this document were developed with this goal in mind, notably the choose of JML as assertion language and the development of JACK and its associated feature. Using those techniques, Java developers should be able to validate their code, or at least to get a good assurance on its correctness.

This document is organized as follows, the next chapter introduce the assertion language JML, chapter 3 describes the JACK tool with its extension feature, chapter 4 presents some evaluations done with the tools and the last chapter concludes.

Chapter 2

Java Validation

This chapter gives an overview of the JML language and the tools that have been developed to deal with the it.

2.1 Java and JavaCard

Formal validation of Java programs is a growing research field. As Java has become a reference language, many technologies are emerging to help Java program validation. Java can also be considered as a good support for formal techniques, as it has precise semantics [18].

JavaCard is a popular programming language for multiple application smart cards. According to the JavaCard Forum [1], which involves key players in the field of smart cards, including smart card manufacturers and banks, the JavaCard language has two important features that make it the ideal choice for smart cards:

- JavaCard programs are written in a subset of Java, using the JavaCard APIs (Application Programming Interfaces). JavaCard developers can therefore benefit from the well-established Java technology;
- the JavaCard security model enables multiple applications to coexist on the same card and communicate securely, and in principle, enables new applications to be loaded on the card after its issuance.

Yet recent research has unveiled several problems in the JavaCard security model, most notably with object sharing and the associated mechanism of shareable interfaces. This has emphasized the necessity to develop environments for verifying the security of the JavaCard platform and of JavaCard programs. Thus far JavaCard security (and also Java security) has been studied mainly at two levels:

[1] http://www.javacardforum.org

- platform level: here the goal is to prove safety properties of the language, in particular type safety and properties related to memory management;

- application level: here the goal is to prove that a specific program obeys a given property, and in particular that it satisfies a security policy, for example based on information flow.

We are focusing at the application level, developing tools and methodologies based on JML to reach this goal.

2.2 JML

JML [24, 25], the "Java Modeling Language", is a behavioral interface specification language for Java; that is, it specifies both the behavior and the syntactic interface of Java code. The syntactic interface of a Java class or interface consists of its method signatures, the names and types of its fields, etc. This is what is commonly meant by an application programming interface (API). The behavior of such an API can be precisely documented in JML annotations; these describe the intended way that programmers should use the API. In terms of behavior, JML can detail, for example, the preconditions and postconditions for methods as well as class invariants.

An important goal for the design of JML is that it should be easily understandable by Java programmers. This is achieved by staying as close as possible to Java syntax and semantics. Another important design goal is that JML *not* impose any particular design method on users; instead, JML should be able to document Java programs designed in any manner [25].

JML uses Java's expression syntax in assertions, thus JML's notation is easy for programmers to learn. Because JML supports quantifiers such as `\forall` and `\exists`, and because JML allows "model" (i.e., specification-only) fields and methods, specifications can easily be made precise and complete. JML assertions are written as special annotation comments in Java code, so that they are ignored by Java compilers but can be used by tools that support JML. Within annotation comments JML extends the Java syntax with several keywords. It also extends Java's expression syntax with several operators. The central ingredients of a JML specification are preconditions (given in `requires` clauses), postconditions (given in `ensures` clauses), and (class and interface) invariants. These are all expressed as boolean expressions in JML's extension to Java's expression syntax. In addition to "normal" postconditions, the language also supports "exceptional" postconditions, specified in `signals` clauses. These can be used to specify what must be true when a method throws an exception.

2.3 The JML tool suite

Since JML specifications are meant to be read and written by ordinary Java programmers, it is important to support the conventional ways that these programmers create and use documentation. Consequently, the `jmldoc` tool produces browsable HTML pages containing both the API and the specifications for Java code, in the style of pages generated by Javadoc [17].

The JML compiler (`jmlc`), developed at Iowa State University, is an extension to a Java compiler and compiles Java programs annotated with JML specifications into Java bytecode [9, 10]. The compiled bytecode includes runtime assertion checking instructions that check JML specifications such as preconditions, normal and exceptional postconditions, invariants, and history constraints. The execution of such assertion checks is transparent in that, unless an assertion is violated, and except for performance measures (time and space), the behavior of the original program is unchanged. The transparency of runtime assertion checking is guaranteed, as JML assertions are not allowed to have any side-effects [26].

2.4 ESC/Java2

ESC/Java2 tool [15], originally developed at Compaq Research, performs what is called "extended static checking" [11, 28], compile-time checking that goes well beyond type checking. It can check relatively simple assertions and can check for certain kinds of common errors in Java code, such as dereferencing `null`, indexing an array outside its bounds, or casting a reference to an impermissible type. ESC/Java2 supports a subset of JML and also checks the consistency between the code and the given JML annotations. The user's interaction with ESC/Java2 is quite similar to the interaction with the compiler's type checker: the user includes JML annotations in the code and runs the tool, and the tool responds with a list of possible errors in the program.

JML annotations affect ESC/Java2 in two ways. First, the given JML annotations help ESC/Java2 suppress spurious warning messages. Second, annotations make ESC/-Java2 do additional checks. In these two ways, the use of JML annotations enables ESC/-Java2 to produce warnings not at the source locations where errors manifest themselves at runtime, but at the source locations where the errors are committed.

Chapter 3

Tools and Methodologies

This chapter describes some tools that have been developed during the Inspired project, that can be handled to validate Java applications and methodologies that can be applied depending on different validation purposes. All those tools are part a Java application validation workshop called JACK.

The first section is a general presentation of the JACK tool, section 2 introduces a JACK extension allowing to verify with some automation some security properties, section 3 presents an associated tool allowing to describe security properties with automata, section 4 is about the proof facilities in JACK, section 5 describes Jack features at bytecode level and section 6 concludes with some perspectives in Java application validation tool development.

3.1 JACK

This section presents the Java Applet Correctness Kit (or `JACK`). This tool, already briefly described in [7, 6], is a formal tool that allows one to prove properties on Java programs using the Java Modeling Language [25] (JML). It generates proof obligations allowing to prove that the Java code conforms to its JML specification. The lemmas are translated into an internal formula language called JPOL. Then JPOL verification conditions are translated into different prover language, namely Coq, PVS, Simplify and the B language [1], allowing to use the automatic provers Simplify and the provers developed within the B method and interactive provers like the Coq proof assistant, PVS or Click'n'Prove .

But the tool is not yet another lemma generator for Java, since it also provides a lemma viewer integrated in the eclipse IDE[1]. This allows to hide the formalisms used behind a graphical interface. Lemmas are presented to users in a way they can understand them easier, by using the Java syntax and highlighting code portions to help the understanding. Using `JACK`, one does not have to learn a formal language to be confident

[1] `http://www.eclipse.org`

on code correctness.

The remainder of the section is organized as follow. Subsection 3.1.1 presents the architecture and the main principles of the tool we have developed. Subsection 3.1.2 presents the Java/Jack Proof Obligation Language (JPOL). Subsection 3.1.3 describes more precisely the innovative parts of the tool and explains why we consider it as accessible to any developers.

3.1.1 Foundations

The main design goals were the following:

- it should provide an easy accessible user interface, that enables average Java programmers to use the tool without too much difficulties. This interface is described section 3.1.3;
- it should provide a high degree of automation, so that most proof obligations can be discharged without user interaction. Only in this way, the tool can be effectively used by non-expert users, which is necessary if we want that formal methods will ever be used in industry.
- it should provide high correctness assurance: at the moment the prover says that a certain proof obligation is satisfied, it should be possible to trust this without any reservation. Nevertheless the tool is not formally developed. It implements, in Java, a weakest precondition calculus that generates lemmas without user interaction. We cannot prove that those lemmas are necessary and sufficient to ensure the correctness of the applet but the tool is designed in this way;
- it should be independent of any particular prover, so that if the use of a particular prover is required (for example by a certification institute) it is relatively easy to adapt the tool accordingly.

This section presents the tool architecture and its principles.

Architecture

Figure 3.1 presents an overview of the `JACK` architecture. `JACK` consists of two parts: a converter (a lemma generator) from Java source annotated with JML into JPOL lemmas, and a viewer that allows developers to understand the generated lemmas. The viewer is integrated in an IDE and is described more precisely in section 3.1.3. This part focuses on the converter.

The `JACK` converter converts a Java class into a JPOL model and allows to prove properties. The initial goal was to prove properties on source files written with the Java

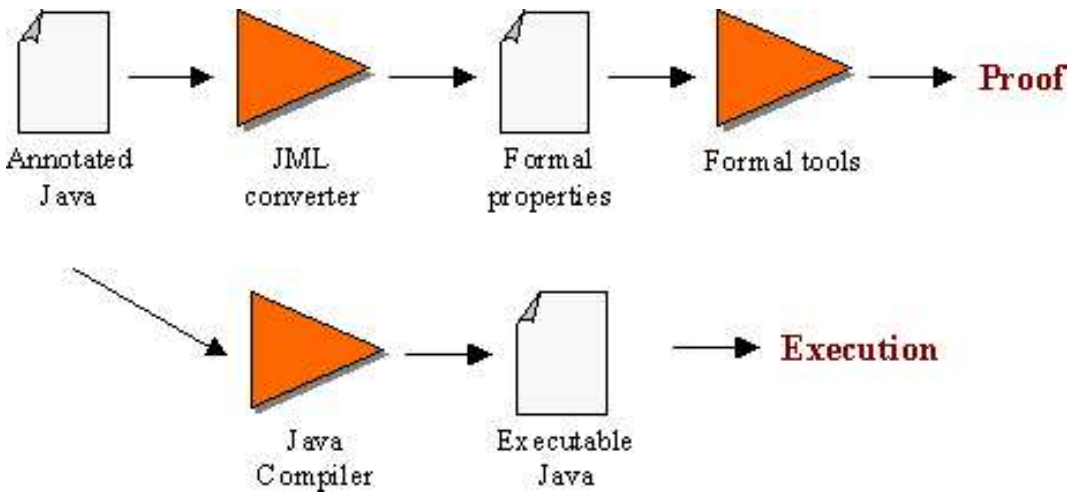

Figure 3.1: JACK ARCHITECTURE

language. To reach this goal, one has to know how to "translate" a Java source file in formal lemmas.

The JML annotations are Java boolean expressions without side effects. Thus, they are easily translated in logical formulas: Java operators are translated into functions. For example, shift left (<<) is translated into a function associating an integer to a pair of integer. From those translated annotations and the methods code, lemmas can be generated automatically.

From the start, taking into account experiences in lemma generation for B machines, we have implemented a Weakest Precondition (WP) calculus to automate lemma generation. Huisman, in [20], presents how the classical Hoare logic can be completed to allow the generation of lemmas in the context of Java. The Java statements contain different features like control-flow breaks. So, the classical WP calculus should be completed to deal with them.

Moreover, JML should be lightly upgraded to allow fully automated proof obligation generation. Notably, to automate lemma generation for the loops, we have had to extend the JML language with new keywords: `loop_modifies`. The `loop_modifies` keyword allows us to declare the variables modified in the body of the loop, as it is done for the methods. During the WP calculus, it is necessary to universally quantify the loop invariant with those variables, and since they cannot be automatically calculated, one has to specify them.

The two main drawbacks of the WP calculus are the loss of information and potential exponential explosion. After lemmas have been generated, it is often difficult to understand from which part of the code they are derived. To bypass this issue, program flow information is associated to each lemma. This information is used in the viewer to associate an execution path to each lemma. This feature is described in the next section.

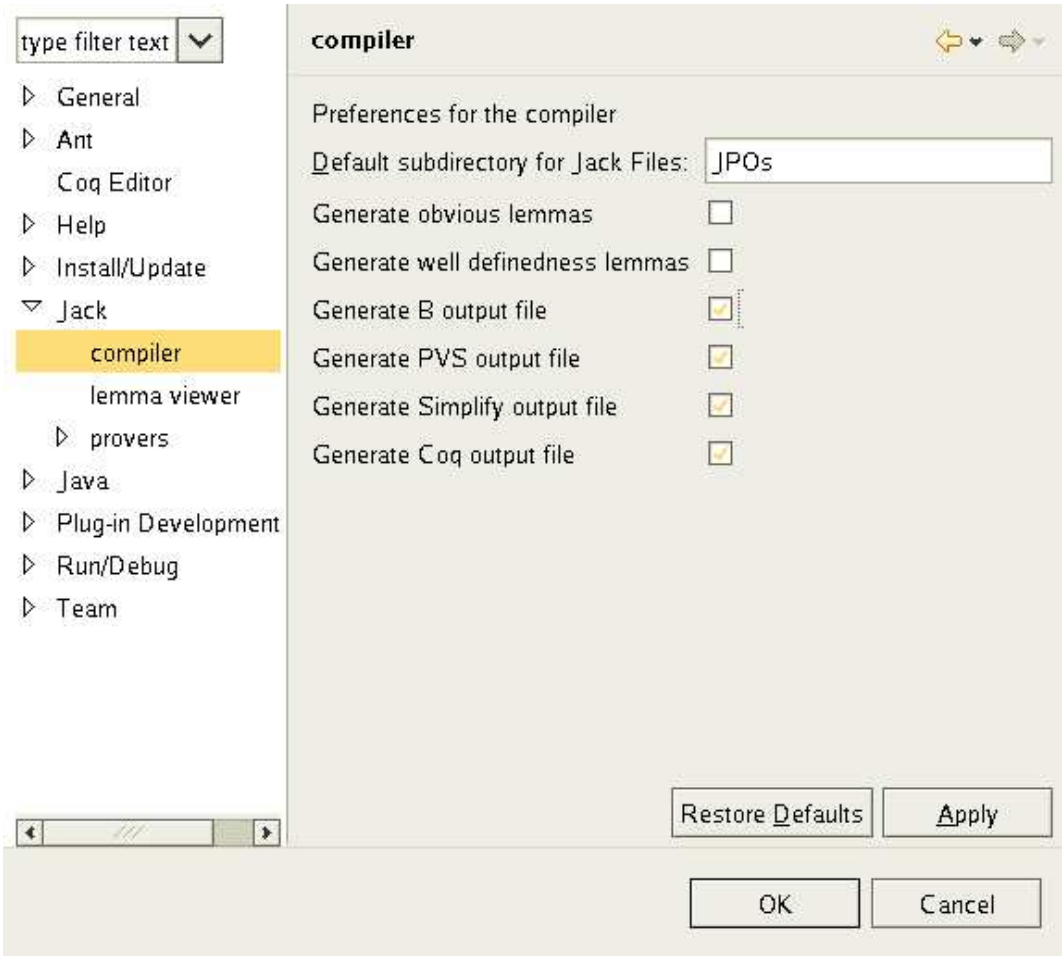

Figure 3.2: JACK COMPILER PREFERENCES PAGE

Exponential explosion remains a problem. Different solutions exist to avoid it. As the WP calculus can be considered as a brute force concept, trying to expand all the path of the methods, solutions are always based on interaction to reduce this brute force by introducing intelligence in the process.

A simple solution is to require users interaction during lemma generation in order to cut unsatisfiable branches. Rather than introducing interaction during generation, another solution is to allow to add special annotations in the source code to introduce formulas that are taken into account at generation to simplify the lemmas. The solution adopted in `JACK` is to allow to specify blocks. An exponential explosion usually occurs in a method with many sequenced branched statement (`if, switch`, etc.) Such methods usually perform different distinct sequenced treatments. Figure 3.3 presents the skeleton of such a method. Specifying a block (here the second part of the method) allows to cut proof obligation generation. This corresponds, in fact, to the simulation of a method call.

```
m() {
   ⋮
   if () { ... }
   else { ... }
   ⋮
   /*@ modifies variables
     @ ensures property
     @*/ {
      ⋮
      if () { ... }
      else { ... }
      ⋮
    }
   }
```

Figure 3.3: Specified block

With those extensions to the JML language, we are able to obtain a fully automated proof obligation generation. That is the first step to reach user approval. The second one is to propose an access to those lemmas in a "Java style", this is described in the next section.

3.1.2 Jack Proof Obligation Language

The Java/Jack Proof Obligation Language is an internal language used in Jack to represent verification conditions issued from the weakest precondition calculus. It can be considered

as a melting-pot language based on first order logic with some specific features like:

- some basic set theory constructions (coming from the B notation)
- some JML keywords
- some Java constructions

The language is typed but has no specific syntax. It corresponds to an internal representation in the tool. Its semantic is given by the translation into the different theorem provers. Adding a theorem prover in JACK corresponds mainly to convert expressions of the JPOL language into the specific theorem prover language.

3.1.3 User Interface

JML has the advantage of being a language that can be rapidly and easily learned and used by developers. One can consider that using a prover is not so easy. Nevertheless formal activities like modeling and proving should not be reserved to experts. To demonstrate this concept, we provide a prover interface understandable to non-experts in formal methods.

In order to simplify the modeling activity with the JML language, our interface requirements are:

- to be integrated with other tools used by developers, and
- not to require the developer to use a mathematical formalism, but hide the mathematical formalism under a "Java" view.

Compared to other formal tools using the JML language, the efforts on the user interface and integration within the development environment is probably the main strength of `JACK`, as is the fact that the underlying mathematical formalism is not exposed to the user.

Integration in developers environment

Java developers are used to develop using integrated development environments (IDE). Those IDEs provide many features useful during the development process. Integrating the tool in such IDEs allows the user to work in a familiar environment. This leads both to better acceptance of the tool, and to a reduced learning curve. Currently, `JACK` is integrated within the eclipse IDE. It could however be ported to other IDEs, and a standalone version that does not require an IDE also exists.

Another constraint has to be taken into account to obtain developer agreement: it is the tool's responsiveness. The tool has to be used interactively, with a debugger spirit: it should not require the developer to wait for a long time. Lemma generation takes, in

realistic examples, less than one minute. Nevertheless, the automatic proof of lemmas is not such a reactive activity. Thus, the tool provides a feature that allows to schedule proof tasks in order to optimize proof time (see paragraph 3.1.4).

Lemma Viewer

One of the most important points of `JACK` is that it does not require developers to learn a mathematical language. Although lemmas are generated, those lemmas are not directly displayed to the user.

Instead, we provide the user with a graphical view (Figure 3.5) of the lemma. The viewer displays

- information concerning the current proof status;
- the class methods with their lemmas;
- the source code;
- and the currently selected lemma (goals and hypotheses with JPOL and prover language translations).

Within a method, each execution path corresponds to a case. Possibly, several lemmas are associated to each case. When a case is selected, the corresponding execution path is highlighted. When a lemma is selected, its views are displayed.

Path highlighting The source code of the program considered is displayed, and the path within the program that leads to the generated proof obligation is highlighted.

Different highlighting colors are used to represent this path:

- green indicates that the corresponding instruction has been executed normally;
- blue indicates that the corresponding instruction has been executed normally, and that additional information is available. For instance, the condition of an `if` construct will usually be displayed in blue with additional information indicating if the condition has been considered as true or false;
- red indicates that the corresponding instruction was supposed to raise an exception when it has been executed in the case considered. Additional information are also provided indicating the exception that has been raised.

The part of the specification (invariant or post-condition) that is involved in the current lemma is also highlighted. Highlighting the part of the source code involved in the proof obligation allows to quickly understand the proof obligation, and allows the user to treat the proof obligations as execution scenarios of the program.

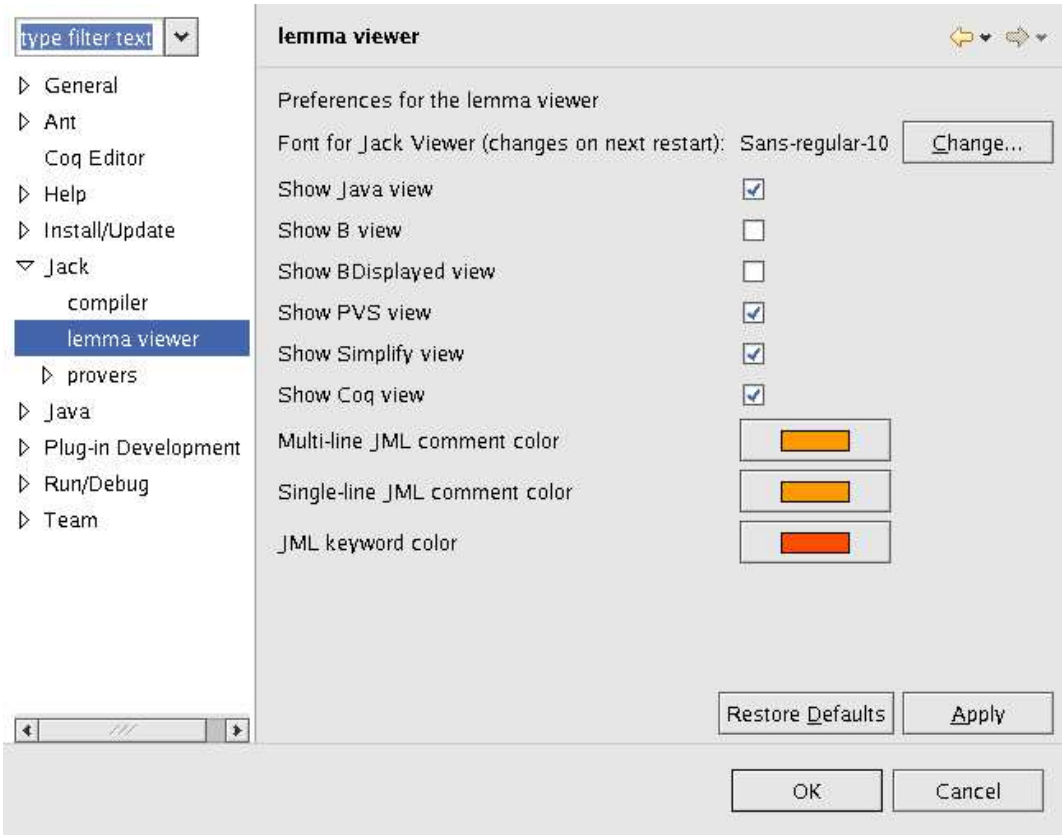

Figure 3.4: JACK LEMMA VIEWER PREFERENCES PAGE

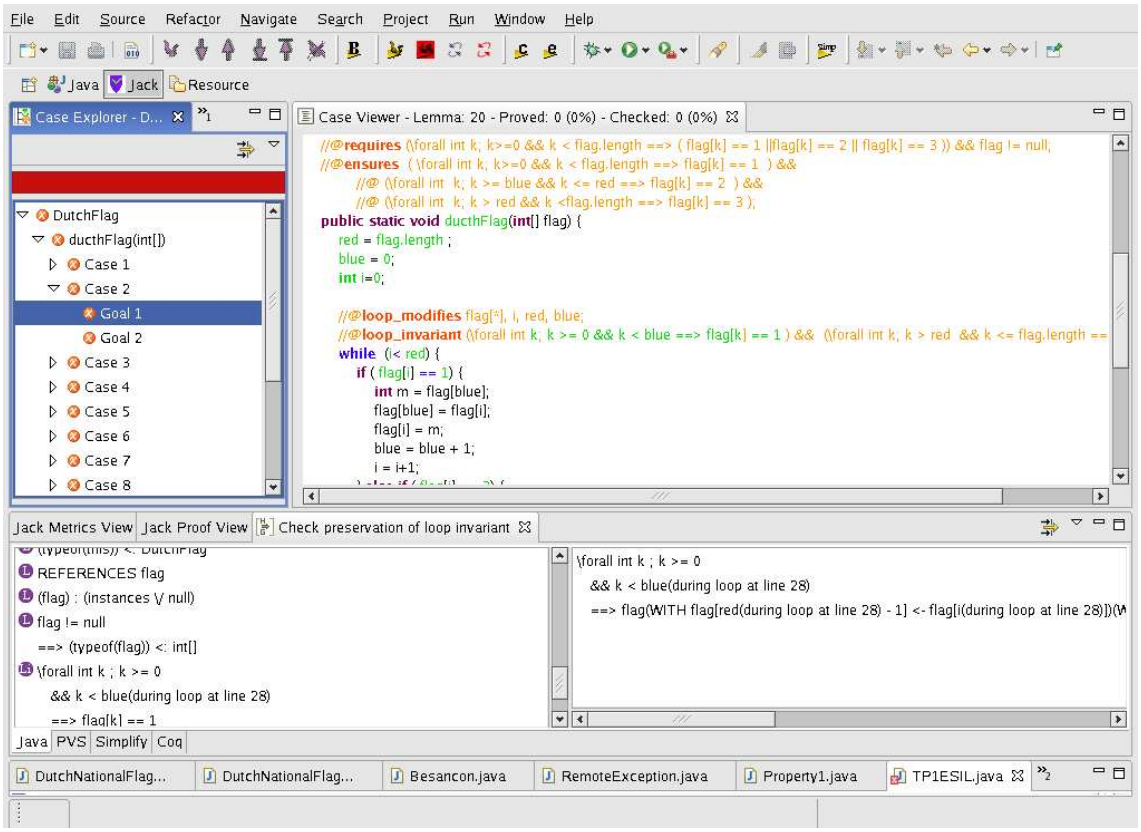

Figure 3.5: Viewer integrated in eclipse

Java presentation of lemmas The hypothesis and goals of the current lemma are also displayed. As the conversion mechanism into provers language may be hard to follow, especially by non-experts, the internal representation used by the tool is used to present the hypothesis and goals in a Java representation. That is, all the variables are displayed using the Java dotted notation, and the Java operators are used instead of their corresponding function.

However, such a translation may be more complicated when operators that have no Java or JML equivalent constructs are used.

However, although the Java view is able to handle some internal representation constructs that do not have direct Java or JML equivalent constructs, there are still constructs that cannot be translated, and for which a Java notation is hard to define. For instance some set operators cannot be translated in a generic way.

3.1.4 Support for verification

Apart from displaying the generated proof obligations, `JACK` also provides support for validating those proof, as detailed hereafter.

Support for automatic proof A point that should not be taken lightly is the time taken by automatic proof: generating proof obligations for industrial size applications will generate thousands of proof obligations.

Typically, those proofs can be quite lengthy, and it is necessary that the user is not obliged to wait for proofs to finish.

To achieve this, `JACK` provides an independent proof view, where files can be queued in order to be submitted to the prover. Thus, the proofs are performed as soon as possible, possibly during the night, allowing the user to focus on cases inspection.

Support for interactive proof Although the automatic prover allows discharging many proof obligations, it cannot discharge all the proof obligations. Thus, the remaining proof obligations have to be verified manually.

Currently, developers are not supposed to handle this task, but to delegate it to a team of experts that would perform the proofs using the interactive prover of the `Atelier B` tool, emacs with PVS or the Coq editor.

Checking proof obligations Additionally to the "*proved*" and "*unproved*" states, `JACK` can also differentiate "*checked*" proof obligations. Checked proof obligations correspond to proof obligations that are not formally proved, but have been manually verified.

Checking a proof obligations is performed by the user to indicate that he has read and understood the proof obligation and has confidence that it is correct. Although the

checked state provides no formal guarantee on the correctness of the proof obligations, it still provides valuable information on the state of a project.

The checked state of the proof obligations can be used in different ways:

- To flag cases as already seen in order to start an interactive proof only if we are pretty sure that the cases are correct, and

- In some cases, when a full correctness assurance of the program is not required, we may accept that not all the proof obligations are formally proved. In that case, it may however, be required that all the proof obligations have been checked.

3.2 Security property propagation

While JML is easily accessible to Java developers and tools exist to manage the annotations, actually writing the specifications of a Java application is labor-intensive and error-prone, as it is easy to forget some annotations. There exist tools which assist in writing these annotations, *e.g.* Daikon [13] and Houdini [14] 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 synthesize annotations from realistic security policies.

We describe here, a method that, given a security policy, automatically annotates a Java (Card) application, in such a way that if the application respects the annotations then it also respects the security policy. The generation of annotations proceeds in two phases: synthesizing and weaving.

1. Based on the security policy we *synthesize* core annotations, specifying the behavior of the methods directly involved.

2. Next we propagate these annotations to all methods directly or indirectly invoking the methods that form the core of the security policy, thus *weaving* the security policy throughout the application.

To show the usefulness of our approach, we applied the algorithm to several realistic examples of smart card applications. When doing this, we actually found violations against the security policies documented for some of these applications.

This section is organized as follows. Subsection 3.2.1 introduces several typical high-level security properties. Next, Subsection 3.2.2 presents the process to weave these properties throughout applications. Subsequently, Subsection 3.2.3 discusses the application of our method to realistic examples.

3.2.1 High-level Security Properties for Applets

The properties that we consider can be divided in several groups, related to different aspects of smart cards. First of all there are properties dealing with the so-called *applet life cycle*, describing the different phases that an applet can be in. Many actions can only be performed when an applet is in a certain phase. Second, there are properties dealing with the transaction mechanism, the Java Card solution for having atomic updates. Further there are properties restricting the kind of exceptions that can occur, and finally, we consider properties dealing with access control, limiting the possible interactions between different applications. For each group we present some example properties. For all these properties encodings into JML annotations exist.

Applet life cycle A typical applet life cycle defines phases as *loading*, *installation*, *personalization*, *selectable*, *blocked* and *dead* (see *e.g.* [33]). Each phase corresponds to a different moment in the applet's life. First an applet is loaded on the card, then it is properly installed and registered with the Java Card Runtime Environment. Next the card is personalized, *i.e.* all information about the card owner, permissions, keys *etc.* is stored. After this, the applet is selectable, which means that it can be repeatedly selected, executed, and deselected. However, if a serious error occurs, for example there have been too many attempts to verify a pin code, the card can get blocked or even become dead. From the latter state, no recovery is possible.

In many of these phases, restrictions apply on who can perform actions, or on which actions can be performed. These restrictions give rise to different security properties, to be obeyed by the applet.

Authenticated initialization Loading, installing and personalizing the applet can only be done by an authenticated authority.

Authenticated unblocking When the card is blocked, only an authenticated authority can execute commands and possibly unblock it.

Single personalization An applet can be personalized only once.

Atomicity 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 synchronized 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, several security properties can be specified.

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 property ensures that the `commitTransaction` will always be executed. If the exception is not caught, the `commitTransaction` would be ignored and the transaction would not be finished. The last property excludes pin verification within a transaction. If this would be allowed, one could abort the transaction every time a wrong pin code has been entered. As this rolls 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.

Exceptions Raising an exception at the top level can reveal information about the behavior of the application and in principle it should be forbidden. However, sometimes it is necessary to pass on information about a problem that occurred. Therefore, the Java Card standard defines so-called ISO exceptions, where a pre-defined status word explains the problem encountered. These exceptions are the only exceptions that may be visible at top-level; all other exceptions should be caught within the application.

Only ISO exceptions at top-level No exception should be visible at top-level, except ISO exceptions.

Access control Another feature of Java Card is an isolation mechanism between applications: the firewall. The firewall ensures that several applications can securely co-exist on the same card, while managing limited collaboration between them: classes and interfaces defined in the same package can freely access each other, while external classes can only be accessed via explicitly shared interfaces. Inter-application communication via shareable interfaces should only take place when the applet is selectable, in all other phases of the applet life cycle only authenticated authorities are allowed to access the applet.

Only selectable applications shareable An application is accessible via a shareable interface only if it is selectable.

3.2.2 Automatic Verification of Security Properties

As explained above, we are interested in the verification of high-level security properties that are not directly related to a single method or class, but that guarantee the overall well-functioning of an application. Writing appropriate JML annotations for such properties

is tedious and error-prone, as they have to be spread all over the application. Therefore, we propose a way to construct such annotations automatically. First we synthesize core-annotations for methods directly involved in the property. For example, when specifying that no nested transactions are allowed, we annotate the methods `beginTransaction`, `commitTransaction` and `abortTransaction`. Subsequently, we propagate the necessary annotations to all methods (directly or indirectly) invoking these core-methods. The generated annotations are sufficient to respect the security properties, *i.e.* if the applet does not violate the annotations, it respects the corresponding high-level security property.

Whether the applet respects its annotations can be established with JACK [6]. Since for most security properties the annotations are relatively simple—but there are many—it is important that these verifications are done automatically, without any user interaction. The results in Section 3.2.3 show that for the generated annotations all correct proof obligations can indeed be automatically discharged.

Architecture

Figure 3.6 shows the general architecture of the tool set for verifying high-level security properties. Our annotation generator can be used as a front-end for any tool accepting JML-annotated Java (Card) applications. As input we have a security property and a Java Card applet. The output is a JML Abstract Syntax Tree (AST), using the format as defined for the standard JML parser. When pretty-printed, this AST corresponds to a JML-annotated Java file. From this annotated file, JACK generates appropriate proof obligations to check whether the applet respects the security property.

Automatic Generation of Annotations

Section 3.2.3 presents example core-annotations for some of the security properties presented in Section 3.2.1, here we focus on the weaving phase, *i.e.* how the core-annotations are propagated throughout the applet. We define functions mod, pre, post and excpost, propagating assignable clauses, preconditions, postconditions and exceptional postconditions, respectively. These functions have been defined and implemented for the full Java Card language, but to present our ideas, we only give the definitions for a representative subset of statements: statement composition, method calls, conditional and `try-catch` statements and special set-annotations. We assume the existence of domains MethName of method names, Stmt of Java Card statements, Expr of Java Card expressions, and Var of static ghost variables, and functions call and body, denoting a method call and body, respectively.

All functions are defined as mutual recursive functions on method names, statements and expressions. When a method call is encountered, the implementation will check whether annotations already have been generated for this method (either by synthesizing or weaving). If not it will recursively generate appropriate annotations. Java Card applets typically do not contain (mutually) recursive method calls, therefore this

does not cause any problems. Generating appropriate annotations for recursive methods would require more care (and in general it might not be possible to do without any user interaction).

3.2.3 Results

For several realistic examples of Java Card applications, we checked whether they respect the security properties presented in Section 3.2.1, and actually found some violations. This section presents these results, focusing on the atomicity properties.

Core-annotations for Atomicity Properties

The core-annotations related to the atomicity properties specify the methods related to the transaction mechanism declared in class `JCSystem` of the Java Card API. As explained above, a static ghost variable `TRANS` is used to keep track of whether there is a transaction in progress.

To check for the absence of uncaught exceptions inside transactions, we use a special feature of JACK, namely pre- and postcondition annotations for statement blocks (as presented in [6]). Block annotations are similar to method specifications. The propagation algorithm is adapted, so that it not only generates annotations for methods, but also for designated blocks. As core-annotation, we add the following annotation for `commitTransaction`.

```
/*@ exsures (Exception) TRANS == 0; @*/
public static native void commitTransaction()
                          throws TransactionException;
```

This specifies that exceptions only can occur if no transaction is in progress. Propagating these annotations to statement blocks ending with a commit guarantees that if exceptions are thrown, they have to be caught within the transaction.

Finally, in order to check that only a bounded number of retries of pin-verification is possible, we annotate the method `check` (declared in the interface `Pin` in the standard Java Card API) with a precondition, requiring that no transaction is in progress.

```
/*@ requires TRANS == 0; @*/
public boolean check(byte[] pin, short offset, byte length);
```

Checking the Atomicity Properties

As mentioned above, we tested our method on realistic examples of industrial smart card applications, including the so-called Demoney case study, developed as a research pro-

totype by Trusted Logic[2], and the PACAP case study[3], 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. All proof obligations generated *w.r.t.* these properties are trivial and can be discharged immediately. However, to emphasize once more the usefulness of having a tool for generating annotations, in the PACAP case study we encountered cases where a single transaction gave rise to twenty-three annotations in five different classes. When writing these annotations manually, it is very easy to forget some of them.

Finally, in the PACAP application we found transactions containing uncaught exceptions. Consider for example the following code fragment.

```
void appExchangeCurrency(...) {
  ...
  /*@ exsures (Exception) TRANS == 0; @*/
  { ...
  JCSystem.beginTransaction();
  try {balance.setValue(decimal2); ...}
  catch (DecimalException e) {
    ISOException.throwIt(PurseApplet.DECIMAL_OVERFLOW); }
  JCSystem.commitTransaction();
  } ... }
```

The method `setValue` that is called can actually throw a decimal exception, which would lead to throwing an ISO exception, and the transaction would not be committed. This clearly violates the security policy as described in Section 3.2.1. After propagating the core-annotations, and computing the appropriate proof obligations, this violation is found automatically, without any problems.

3.2.4 Properties as automata

A step forward to lower the gap between specification and annotations is to describe properties at a more abstract level. This move towards high level specification is deeply facilitated by the frequent use of tools such as UML to specify software components.

[2] `http://www.trusted-logic.fr`
[3] `http://www.gemplus.com/smart/r_d/publications/case-study`

The Automaton model

A minimal set of constraint can be defined concerning the choice of the abstract model with the hope of keeping most of JML's expressiveness. First of all, the chosen model needs to be able to generate JML annotation for native methods. This requirement is mainly due to the fact that Java Card API is not provided together with JML specification. As a result, the model being chosen must be able to fit with both native and non-native methods. Secondly, the abstract model should consider methods' sequences as well as their pre and postconditions : this level of description surely is not only the most adequate to connect implementation to its high level specification, but also very crucial to smart verification. Furthermore, this choice is of great importance to connect the present work with the existing propagation tool. Finally, all possible improvement such as the support of invariants would be definitely a plus to keep as much as possible of the JML expressiveness. For many reasons, the choice of finite state machine as a model to describe these properties is about to fulfill the previous expectations. FSM have been historically employed to model check software execution. Thereof, it is possible to assert that it is capable of specifying valid and invalid sequences in an intuitive way to people of the verification field. Nevertheless, the semantic of this automaton would require to be adapted to the expression of properties. It will be in particular compulsory to make an automaton stick to a property, by defining what could be the properties' states and what could make it switch from one to another. However, automatons make trivial the possibility to express pre and postcondition as they can be seen as methods' use cases, which is possible to describe through automatons' conditional evolution. As a consequence, the model used for this present study will be the FSM model.

Translating automatons to Properties

Starting from the use of FSM, the initial problem was to correlate the semantic of states and transitions with a program execution to characterize properties. On one hand, the state should be characterized by a unique set of properties from which a given set of evolution is possible. Hence, a program state could be defined by a set of values attached to program variables, or possibly by method's status (i.e. active state would represent the method currently executed). On the other hand, the evolution between those states is expressible with the automaton model through guards, update, and message passing. Once again, several possibilities are acceptable. In the case where the active state represents the method currently executing, transitions might be use to express pre and postconditions through guards and even specify some entry/exit action in the update field. Nevertheless, it is as well acceptable to assert that program's evolution is conditioned by the sequent call and returns of method, so that transition should be attached to methods, and pre and postconditions could be specified as state invariants.

Even though several formalisms would be suitable, many of them could be discarded due to the limited nature of the properties expressible. First, correlating the active state with the method currently executing happened to be a bad idea. Such a representation

imposes transition to define the pre and postcondition with the meaning that a method would neither be executed without respecting the preconditions of the incoming transitions nor terminate without fulfilling the postcondition specified as guard. For a unique transition plays both the roles of pre and post condition, choosing this modeling scheme would be perfectly appropriate to for expressing exact sequences of method call for which postcondition stands also as the precondition of the next method being called. This could be practical for describing completely known sequences which would be the case for protocols specification. However, considering the annotation of the `beginTransaction` and `commitTransaction` methods (which are involved in the transaction process of smart cards) demonstrates in the general case the need to partially specify sequences as methods have to be called in between those two methods. Moreover, this representation is not suitable to express properties dealing with recursion.

As a result, methods will be attached to transition through the definition of method's events. Because transitions are meant to express a logical condition for going from the active state to the next, the FSM model supposes that the transition to take no time for switching. Therefore, events on methods should be define so that to be expressible in transitions. Fortunately, these events appear obvious for they are exactly the one considered in specifying pre and postcondition. First is the method call, which will cause the method to be executed. Second and third are the normal termination invoke by a simple return statement or an exceptional termination triggered by the throw statement. As a consequence, the whole systems evolution will be conditioned by 3 types of event which could stand for any language supporting exceptions.

From what has been defined so far, transitions' role could finally be defined to carrying method's contract and the contribution of states to the model could be clarified. Yet, only two possibilities are offered to specify the contract: either states or transitions have to carry the pre and postconditions. The choice of state invariant to carry this information was discarded right away for the simple reason that a unique description would again stand for sequent post and preconditions. Guards appeared to be more likely to hold the contract for it would link methods' events to their triggering conditions. In other words, taking a transition would mean that the associated method call or return was done with respect to the pre or postcondition. This solution presents the huge advantage of keeping state free to specify invariants in accordance with the meaning a user would give to it.

3.3 At bytecode level

We propose a bytecode verification framework with the following components: a bytecode specification language, a compiler from source program annotations into bytecode annotations and a verification condition generator over Java bytecode.

In a client-producer scenario, these features bring to the producer means to supply the sufficient specification information which will allow the client to establish trust in

the code, especially when the client policy is potentially complex and a fully automatic specification inference will fail. On the other hand, the client is supplied with a procedure to check the untrusted annotated code.

Our approach is tailored to Java bytecode. The Java technology is widely applied to mobile and embedded components because of its portability across platforms. For instance, its dialect JavaCard is largely used in smart card applications and the J2ME Mobile Information Device Profile (MIDP) finds application in GSM mobile components.

The proposed scheme is composed of several components. We define a bytecode specification language, called BCSL, and supply a compiler from the high level Java specification language JML [27] to BCSL. BCSL supports a JML subset which is expressive enough to specify rich functional properties. The specification is inserted in the class file format in newly defined attributes and thus makes not only the code mobile but also its specification. These class file extensions do not affect the JVM performance. We define a bytecode logic in terms of weakest precondition calculus for the sequential Java bytecode language. The logic gives rules for almost all Java bytecode instructions and supports the Java specific features such as exceptions, references, method calls and subroutines. We have implementations of a verification condition generator based on the weakest precondition calculus and of the JML specification compiler. Both are integrated in the Java Applet Correctness Kit tool (JACK) [6].

The full specifications of the JML compiler, the weakest precondition predicate transformer definition and its proof of correctness can be found in [37].

The remainder of this section is organized as follows: Subsection 3.3.1 reviews scenarios in which the architecture may be appropriate to use; Subsection 3.3.2 presents the bytecode specification language BCSL and the JML compiler; Subsection 3.3.4 discusses the main features of the *wp*(short for weakest precondition calculus); Subsection 3.3.5 discusses the relationship between the verification conditions for JML annotated source and BCSL annotated bytecode.

3.3.1 Applications

The overall objective is to allow a client to trust a code produced by an untrusted code producer. Our approach is especially suitable in cases where the client policy involves non trivial functional or safety requirements and thus, a full automatization of the verification process is impossible. To this end, we propose a PCC technique that exploits the JML compiler and the weakest predicate function presented in the article.

The framework is presented in Fig. 3.7; note that certificates and their checking are not yet implemented and thus are in oblique font.

In the first stage of the process the client provides the functional and (or) security requirements to the producer. The requirements can be in different form:

- Typical functional requirements can be a specified interface describing the applica-

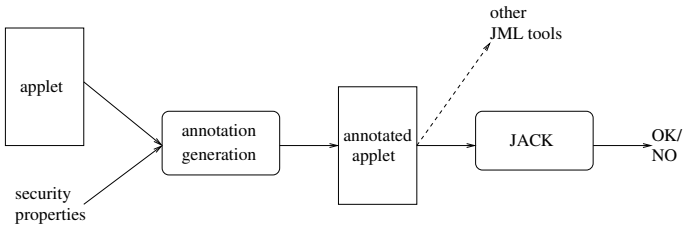

Figure 3.6: Tool set for verifying high-level security properties

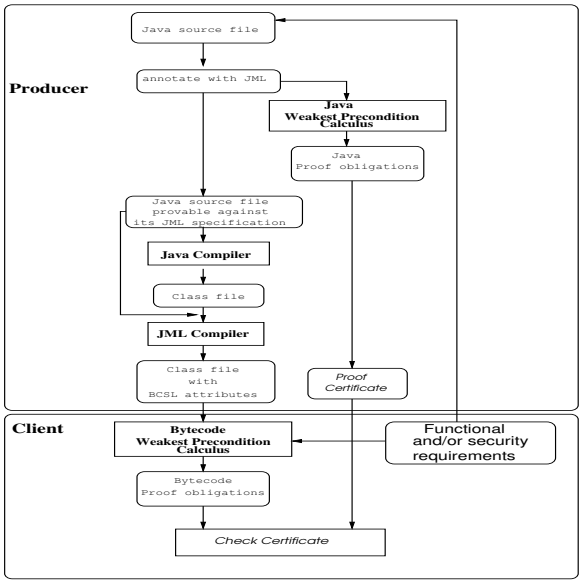

Figure 3.7: The overall architecture for client producer scenarios

tion to be developed. In that case, the client specifies in JML the features that have to be implemented by the code producer.

- Client security requirements can be a restricted access to some method from the API expressed as a finite state machine. For example, suppose that the client API provides transaction management facilities - the API method `open` for opening and method `close` for closing transactions. In this case, a requirement can be for no nested transactions. This means that the methods `open` and `close` can be annotated to ensure that the method `close` should not be called if there is no transaction running and the method `open` should not be called if there is already a running transaction. In this scenario, we can apply results of previous work [36].

Usually, the development process involves annotating the source code with JML specification, generating verification conditions, using proof obligation generator over the source code and discharging proofs which represent the program safety certificate and finally, the producer sends the certificate to the client along with the annotated class files. Yielding certificates over the source code is based on the observation that proof obligations on the source code and non-optimized bytecode respectively are syntactically the same modulo names and basic types. Every Java file of the untrusted code is normally compiled with a Java compiler to obtain a class file. Every class file is extended with user defined attributes that contain the BCSL specification, resulting from the compilation of the JML specification of the corresponding Java source file.

We have extended the Jack tool with a compiler from JML to BCSL and a bytecode verification condition generator. In the next sections, we introduce the BCSL language, the JML compiler and the bytecode *wp*calculus which underlines the bytecode verification condition generator.

3.3.2 Bytecode Specification Language

In this section, we introduce a bytecode specification language which we call BCSL (short for ByteCode Specification Language). BCSL is based on the design principles of JML (Java Modeling Language) [27], which is a behavioral interface specification language following the design by contract approach [3].

In the following, we give the grammar of BCSL and sketch the compiler from JML to BCSL.

Grammar

BCSL corresponds to a representative subset of JML and is expressive enough for most purposes including the description of non trivial functional and security properties.

Specification clauses in BCSL that are taken from JML and inherit their semantics directly from JML include: class specification, i.e. class invariants and history constraints,

method preconditions, normal and exceptional postconditions, method frame conditions (the locations that may be modified by the method), inter method specification, as for instance loop invariants and loop frame conditions(this is not a standard feature of JML but we were inspired for this by the JML extensions in JACK [6]). We also support specification inheritance and thus behavioral subtyping as described in [12]. Most of the Java expressions like field access expressions, local variables, etc can be mentioned in the BCSL specification. BCSL supports the standard JML specification operators as for example, \old$\mathcal{E}$ which is used in method postconditions and designates the value of the expression $\mathcal{E}$ in the prestate of a method, `\result` which stands for the value the method returns if it is not void, `\typeof(`$\mathcal{E}$`)` which stands for type of $\mathcal{E}$ etc.

3.3.3 Compiling JML into BCSL

We now turn to explaining how JML specifications are compiled into user defined attributes for Java class files. Recall that a class file defines a single class or interface and contains information about the class name, interfaces implemented by the class, super class, methods and fields declared in the class and references. The Java Virtual Machine Specification (JVM) [31] mandates that the class file contains data structure usually referred as the **constant_pool** table which is used to construct the runtime constant pool upon class or interface creation. The runtime constant pool serves for loading, linking and resolution of references used in the class. The JVM allows to add to the class file user specific information([31], ch.4.7.1). This is done by defining user specific attributes (their structure is predefined by JVM).

Thus the "JML compiler" [4] compiles the JML source specification into user defined attributes. The compilation process has three stages:

1. Compilation of the Java source file. This can be done by any Java compiler that supplies for every method in the generated class file the **Line_Number_Table** and **Local_Variable_Table** attributes. The presence in the Java class file format of these attribute is optional [31], yet almost all standard non optimizing compilers can generate these data. The **Line_Number_Table** describes the link between the source line and the bytecode of a method. The **Local_Variable_Table** describes the local variables that appear in a method. Those attributes are important for the next phase of the JML compilation.

2. Compilation of the JML specification from the source file and the resulting class file. In this phase, Java and JML source identifiers are linked with their identifiers on bytecode level, namely with the corresponding indexes either from the cp (short for constant pool) or the array of local variables described in the **Local_Variable_Table** attribute. If a field identifier, for which no cp index exists, appears in the JML specification, a new index is added in the cp and the field identifier in question

[4] not to be confused, Gary Leavens also calls his tool jmlc JML compiler, which transforms JML into runtime checks and thus generates input for the jmlrac tool

is compiled to the new cp index. It is also in this phase that the specification parts like the loop invariants and the assertions which should hold at a certain point in the source program must be associated to the respective program point on bytecode level. The specification is compiled in binary form using tags in the standard way. The compilation of an expression is a tag followed by the compilation of its subexpressions.

Another interesting point in this stage of the JML compilation is how the type differences on source and bytecode level are treated. The JVM does not provide a direct support for integral types like byte, short, char, neither for boolean. Those types are rather encoded as integers in the bytecode. The JML compiler performs transformation on specifications that involve Java boolean values and variables.

3. add the result of the compiled specifications components in newly defined attributes in the class file. For example, the specifications of all the loops in a method are compiled to a unique method attribute whose syntax is given in Fig. 3.8. This attribute is an array of data structures each describing a single loop from the method source code. More precisely, every element contains information about the instruction where the loop starts as specified in the **Line_Number_Table**, the locations that can be modified in a loop iteration, the invariant associated to this loop and the decreasing expression in case of total correctness,

```
JMLLoop_specification_attribute {
    ...
    {  index;
       modifies_count;
       formula modifies[modifies_count];
       formula invariant;
       expression decreases;
    } loop[loop_count];
}
```

Figure 3.8: Structure of the loop specification attribute

The most problematic part of the specification compilation is the identification of which loop in the source corresponds to which bytecode loop in the control flow graph. To do this, we assume that the control flow graph is reducible (see [2]), i.e. there are no jumps into the middle of the loops from outside; graph reducibility allows to establish the same order between loops in the bytecode and source code level and to find the right places in the bytecode where the loop invariants must hold.

3.3.4 Weakest Precondition Calculus For Java Bytecode

In this section, we define a bytecode logic in terms of a weakest precondition calculus. The proposed weakest precondition *wp* supports all Java bytecode sequential instructions except for floating point arithmetic instructions and 64 bit data (`long` and `double` types), including exceptions, object creation, references and subroutines. The calculus is defined over the method control flow graph and supports BCSL annotation, i.e. bytecode method's specification like preconditions, normal and exceptional postconditions, class invariants, assertions at particular program point among which loop invariants. The verification condition generator applied to a method bytecode generates a proof obligation for every execution path by applying first the weakest predicate transformer to every `return` instruction, `athrow` instruction and end of a loop instruction and then following in a backwards direction the control flow up to reaching the entry point instruction. In an extended version of the present paper [37], we show that the *wp* function is correct.

In Fig. 3.9, we show the *wp* rule for the `Type_load i` instruction. As the example shows the *wp* function takes three arguments: the instruction for which we calculate the precondition, the instruction's postcondition ψ and the exceptional postcondition function ψ^{exc} which for any exception `Exc` and instruction index `ind` returns the corresponding exceptional postcondition $\psi^{exc}(\texttt{Exc}, \texttt{ind})$. One can also notice that the rule involves the stack expressions `ct`(stands for the counter of the method execution stack) and `st( i )` (stands for the element at ind `i` from the stack top). This is because the JVM is stack based and the instructions take their arguments from the method execution stack and put the result on the stack. The *wp* rule for `Type_load i` increments the stack counter `ct` and loads on the stack top the contents of the local variable `lv[i]`.

$$
\begin{array}{l}
wp(\texttt{Type\_load}\ \ \texttt{i},\ \psi,\ \psi^{exc}) = \\
\ \psi[\texttt{ct} \leftarrow \texttt{ct} + 1][\texttt{st(ct+1)} \leftarrow \texttt{lv[i]}]
\end{array}
$$

$$
\begin{array}{l}
wp(\texttt{putField Cl.f},\ \psi,\ \psi^{exc}) = \\
\ \texttt{st(ct-1)} \neq \texttt{null} \Rightarrow \psi\ \begin{array}{l}[\texttt{ct} \leftarrow \texttt{ct} - 2] \\ [\texttt{Cl.f} \leftarrow \texttt{Cl.f} \oplus [\texttt{st(ct-1)} \rightarrow \texttt{st(ct)}]]\end{array} \\
\ \wedge \\
\ \texttt{st(ct-1)} = \texttt{null} \Rightarrow \psi^{exc}(\texttt{NullPointerExc})\ \begin{array}{l}[\texttt{ct} \leftarrow 0] \\ [\texttt{st(0)} \leftarrow \texttt{st(ct)}]\end{array}
\end{array}
$$

Figure 3.9: Examples for bytecode wp rules

In the following, we consider how instance fields, loops exception handling and subroutines are treated. We omit here aspects like method invocation and object creation because of space limitations but a detailed explanation can be found in [37].

Manipulating object fields Instance fields are treated as functions, where the domain of a field `f` declared in the class `Cl` is the set of objects of class `Cl` and its subclasses. We are using function updates when assigning a value to a field reference as, for instance in [4]. In Fig.3.9, we give the *wp* rule for the instruction `putfield Cl.f`, which updates the field `Cl.f`[5] of the object referenced by the reference stored in the stack below the stack top `st(ct-1)` with the value on the stack top `st(ct)`. Note that the rule takes in account the possible exceptional termination of the instruction execution.

Loops Identifying loops on bytecode and source programs is different because of their different nature — the first one lacks while the second has structure. While on source level loops correspond to loop statements, on bytecode level we have to analyze the control flow graph in order to find them. The analysis consists in looking for the backedges in the control flow graph using standard techniques, see [2].

We assume that a method's bytecode is provided with sufficient specification and in particular loop invariants. Under this assumption, we build an abstract control flow graph where the backedges are replaced by the corresponding invariant. We apply the *wp* function over the abstract version of the control flow graph which generates verification conditions for the preservation and initialization of every invariant in the abstraction graph.

Exceptions and Subroutines Exception handlers are treated by identifying the instruction at which the handler compilation starts. The JVM specification mandates that a Java compiler must supply for every method an **Exception_Table** attribute that contains data structures describing the compilation of every implicit (in the presence of subroutines) or explicit exception handler: the instruction at which the compiled exception handler starts, the protected region (its start and end instruction indexes), and the exception type the exception handler protects from. Thus, for every instruction `ins` in method `m` which may terminate exceptionally on exception `Exc` the exceptional function ψ^{exc} returns the *wp* predicate of the exceptional handler protecting `ins` from `Exc` if such a handler exists. Otherwise, ψ^{exc} returns the specified exceptional postcondition for exception `Exc` as specified in the specification of method `m` .

Subroutines are treated by abstract inlining[6]. First, the instructions of every subroutine are identified. To this end, we assume that the code has passed the bytecode verification and that every subroutine terminates with a `ret` instruction(usually, the compilation of subroutines ends with a `ret` instruction but it is not always the case). Thus, by abstract inlining, we mean that whenever the *wp* function is applied to an instruction `jsr ind`, a postcondition ψ and an exceptional postcondition function ψ^{exc}, its precondition $wp^{\texttt{jsr ind}}$ is calculated as follows: the *wp* is applied to the bytecode instructions

[5] `Cl.f` stands for the field `f` declared in class `Cl`

[6] NB: we do not transform the bytecode. It is rather the *wp* function that treats subroutines as if the subroutines were inlined

that represent the subroutine which starts at instruction `ind`, the postcondition ψ and the exceptional postcondition function ψ^{exc}.

3.3.5 Relation between verification conditions on source and bytecode level

We studied the relationship between the source code proof obligations generated by the standard feature of JACK and the bytecode proof obligations generated by our implementation over the corresponding bytecode produced by a non optimizing compiler over the examples given in [22]. The proof obligations were the same modulo names and short, byte and boolean values as well as hypothesis names. The proof obligations on bytecode and source level that we proved interactively in Coq produced proof scripts which were also equal modulo those values and the hypothesis's names. This means that an appropriate encoding of proof obligations on source and bytecode level can be found, where the names of the source and bytecode hypothesis are the same and where the source obligations can be transformed in such a way that the produced proof script for the transformed source proof obligation can be applied to the corresponding one on bytecode level.

The equivalence between source and bytecode proof obligations can be applied to PCC scenarios, as discussed in Section 3.7 in cases where the client policy is complex and a complete automatic certification will not work and thus an interactive generation of the certificate is an adequate solution.

Chapter 4

Evaluations

4.1 Industrial evaluation: Oberthur

4.2 Industrial evaluation: Axalto

4.3 Bytecode Verifier

The tools have also been tested on a bytecode verifier java implementation. A termination proof has been provided. A specific implementation has been coded with on one hand the main loop which remain unchanged whatever the specifications of the virtual machine Java chosen, and other instructions and memory states which depends on selected model.

4.3.1 Implementation and Modelisation

The main loop is in a package which contains abstract classes: the instructions and the states are implemented in a more generic way. The package containing the implementation is composed from the instructions for the standard Java types and of the states of memory typing.

Memory states

The memory states are represented by the State class, which is an abstract class. It does not contain any precise definition of the memory: one has no information on the stack or on the local variables table. The implementation is relatively simple: it is a class which contains a type stack and a table of the types of the local variables. Functions allowing to read simply these structures and to generate verification error in the cases of misuse are defined.

Instructions

The instructions are also represented by an abstract class: the class Instruction. Since in the Kildall algorithm each instruction is associated to a memory state, the Instruction class has a field of the State type. An instruction can also have one or more successors. This relation is represented by a field which is the list of the successors of the instruction. One of the other aspects is the fact that on associate to each instruction a boolean field to determine if it has been modified or not.

Several properties of the bytecode verifier are formalised in this class. First of all one verifies that the successors of the instruction are well included in the others instructions of the program. If these successors pointed towards external instructions, an verification error would be returned.

The others important properties concern the pure function `buildNewState`. This function builds the typing state of the execution of an instruction on the current state. This construction can fail if the instruction tries for example to pop an element when the stack is empty. If it succeeds, a new non null state is build.

Around ten instructions have been implemented: `load` and `blind` for the access to local variables, `push` and `pop` to obtain or put element on the stack, `op1` and `op2` which is two operators who consume both the two top element of the stack and which replaces them by a result of a certain type, `ifle` and `jump` instructions of jump towards another instruction successor, `nop` the instruction which does not do anything and finally `stop` which is an instruction which does not have a successor. These instructions have an associated type in the OperandType class, who can be None, Type1 or Type2. Those are the minimal instructions to have a Java-like program.

The main loop

The main loop is implemented in the Verifier class. It is not an abstract class because it uses the properties of the State and Instruction abstract classes to verify an instruction set on particular states. This class provides two functions, the function `verify` in which the loop is written and the function `check` which verify an instruction.

The tt verify method is the main loop of the bytecode verifier. It contains two nested loops. The internal one is a `for` loop which iterates on the instructions and verify all the quoted instructions (as described in the Kildall algorithm). The termination of the internal loop is easy to prove. The `for` loop executes as many time as there are numbers in the table. The external `while` loop stops the algorithm when no more instruction typing state is modified. This termination is not obvious to prove, especially with JML, since it only allow to prove loop termination by giving an integer variant.

Since the states have to be used to show the algorithm termination, one has to make correspond each state with an integer. Thus at each loop iteration, the integer associated with the state either increase or preserve the same value; and it exists a maximum value.

Classes:	State	Instruction	Verifier
Lines of code:	14	47	66
Lines of annotations:	20	54	81
Proof obligations:	26	129	627
Automatically proved proof obligations:	17	93	112
Average length of a non-automatic proof:	3	6	12

Figure 4.1: Some statistics on proof

4.3.2 Proofs

The first proofs are relatively easy. The State class is proved almost automatically; except for the constructor where it is necessary to break a disjunction (`instance S` $\vee$ `S = null`) to prove the invariants.

The Instruction class has also been relatively easy to prove. A significant number of proof was done automatically (approximately 90 %); then majority of proof could be trivially resolved, except some lemma concerning a loop termination.

Finally the Verifier class was harder to prove. The lemmas were containing too many hypotheses to be automatically proved. Around 500 proof obligations have to be resolved manually. Some of them was obvious and were resolved with quite the same script, but the script cannot be automated. Some of them was complex: the proof script became little large (an average of 30 steps). The lemmas concerning the loop invariant of the verify method and its initialization were the most difficult.

4.4 Low-Footprint Java-to-Native Compilation

Enabling Java on embedded and restrained systems is an important challenge for today's industry and research groups [34]. Java brings features like execution safety and low-footprint program code that make this technology appealing for embedded devices which have obvious memory restrictions, as the success of Java Card witnesses. However, the memory footprint and safety features of Java come at the price of a slower program execution, which can be a problem when the host device already has a limited processing power. As of today, the interest of Java for smart cards is still growing, with next generation operating systems for smart cards that are closer to standard Java systems [23, 19], but runtime performance in still an issue. To improve the runtime performances of Java systems, a common practice is to translate some parts of the program bytecode into native code.

Doing so removes the interpretation layer and improves the execution speed, but

also greatly increases the memory footprint of the program: it is expected that native code is about three to four times the size of its Java counterpart, depending on the target architecture. This is explained by the less-compact form of native instructions, but also by the fact that many safety-checks that are implemented by the virtual machine must be reproduced in the native code. For instance, before dereferencing a pointer, the virtual machine checks whether it is `null` and, if it is, throws a `NullPointerException`. Every time a bytecode that implements such safety-behaviors is compiled into native code, these behaviors must be reproduced as well, leading to an explosion of the code size. Indeed, a large part of the Java bytecode implement these safety mechanisms.

Although the runtime checks are necessary to the safety of the Java virtual machine, they are most of the time used as a protection mechanism against programming errors or malicious code: A runtime exception should be the result of an exceptional, unexpected program behavior and is rarely thrown when executing sane code - doing so is considered poor programming practice. The safety checks are therefore without effect most of the time, and, in the case of native code, uselessly enlarge the code size.

Several studies proposed to factorize these checks or in some case to eliminate them, but none proposed a complete elimination without hazarding the system security. In this paper, we use formal proofs to ensure that run-time checks can never be true into a program, which allows us to completely and safely eliminate them from the generated native code. The programs to optimize are JML-annotated against runtime exceptions and verified by the Java Applet Correctness Kit (JACK [6]). We have been able to remove almost all of the runtime checks on tested programs, and obtained native ARM thumb code which size was comparable to the original bytecode.

4.4.1 Java and Ahead-of-Time Compilation

Compiling Java into native code common on embedded devices. This section gives an overview of the different compilation techniques of Java programs, and points out the issue of runtime exceptions.

Ahead-of-Time & Just-in-Time Compilation

Ahead-of-Time (AOT) compilation is a common way to improve the efficiency of Java programs. It is related to Just-in-Time (JIT) compilation by the fact that both processes take Java bytecode as input and produce native code that the architecture running the virtual machine can directly execute. AOT and JIT compilation differ by the time at which the compilation occurs. JIT compilation is done, as its name states, just-in-time by the virtual machine, and must therefore be performed within a short period of time which leaves little room for optimizations. The output of JIT compilation is machine-language. On the contrary, AOT compilation compiles the Java bytecode way before the program is run, and links the native code with the virtual machine. In other words, it translates non-native methods into native methods (usually C code) prior to the whole

system execution. AOT compilers either compile the Java program entirely, resulting in a 100% native program without a Java interpreter, or can just compile a few important methods. In the latter case, the native code is usually linked with the virtual machine. AOT compilation have no or few time constraints, and can generate optimized code. Moreover, the generated code can take advantage of the C compiler's own optimizations.

JIT compilation in interesting by several points. For instance, there is no prior choice about which methods must be compiled: the virtual machine compiles a method when it appears that doing so is beneficial, e.g. because the method is called often. However, JIT compilation requires embedding a compiler within the virtual machine, which needs resources to work and writable memory to store the compiled methods. Moreover, the compiled methods are present twice in memory: once in bytecode form, and another time in compiled form. While this scheme is efficient for decently-powerful embedded devices such as PDAs, it is inapplicable to very restrained devices like smartcards or sensors. For them, ahead-of-time compilation is usually preferred because it does not require a particular support from the embedded virtual machine outside of the ability to run native methods, and avoids method duplication. AOT compilation has some constraints, too: the compiled methods must be known in advance, and dynamically-loading new native methods is forbidden, or at least very unsafe.

Both JIT and AOT compilers must produce code that exactly mimics the behavior of the Java virtual machine. In particular, the safety checks performed on some bytecode must also be performed in the generated code.

Java Runtime Exceptions

The JVM (Java Virtual Machine) [31] specifies a safe execution environment for Java programs. Contrary to native execution, which does not automatically control the safety of the program's operations, the Java virtual machine ensures that every instruction operates safely. The Java environment may throw predefined runtime exceptions at runtime, like the following ones:

If the JVM detects that executing the next instruction will result in an inconsistency or an illegal memory access, it throws a runtime exception, that may be caught by the current method or by other methods on the current stack. If the exception is not caught, the virtual machine exits. This safe execution mode implies that many checks are made during runtime to detect potential inconsistencies.

Of the 202 bytecodes defined by the Java virtual machine specification, we noticed that 43 require at least one runtime exception check before being executed. While these checks are implicitly performed by the bytecode interpreter in the case of interpreted code, they must explicitly be issued every time such a bytecode is compiled into native code, which leads to a code size explosion. Ishizaki et al. measured that bytecodes requiring runtime checks are frequent in Java programs: for instance, the natively-compiled version of the SPECjvm98 `compress` benchmark has 2964 exception check sites for a size of 23598 bytes. As for the `mpegaudio` benchmark, it weights 38204 bytes and includes

6838 exception sites [21]. The exception check sites therefore make a non-neglectable part of the compiled code.

4.4.2 Optimizing Ahead-of-Time Compiled Java Code

Verifying that a bytecode program does not throw Runtime exceptions using JACK involves several stages:

1. writing the JML specification at the source level of the application, which expresses that no runtime exceptions are thrown.
2. compiling the Java sources and their JML specification[1].
3. generating the verification conditions over the bytecode and its BCSL specification, and proving the verification conditions 4.4.2. During the calculation process of the verification conditions, they are indexed with the index of the instruction in the bytecode array they refer to and the type of specification they prove (e.g. that the proof obligation refers to the exceptional postcondition in case an exception of type `Exc` is thrown when executing the instruction at index `i` in the array of bytecode instructions of a given method). Once the verifications are proved, information about which instructions can be compiled without runtime checks is inserted in user defined attributes of the class file.
4. using these class file attributes in order to optimize the generated native code. When a bytecode that has one or more runtime checks in its semantics is being compiled, the bytecode attribute is checked in order to make sure that the checks are necessary. It indicates that the exceptional condition has been proved to never happen, then the runtime check is not generated.

Our approach benefits from the accurateness of the JML specification and from the bytecode verification condition generator. Performing the verification over the bytecode allows to easily establish a relationship between the proof obligations generated over the bytecode and the bytecode instructions to optimized.

In the rest of this section, we explain in detail all the stages of the optimization procedure.

Methodology for Writing Specification Against Runtime Exception

We now illustrate with an example which annotations must be generated in order to check if a method may throw an exception. Figure 4.2[2] shows a Java method annotated with

[1] the BCSL specification is inserted in user defined attributes in the class file and so does not violate the class file format

[2] although the analysis that we describe is on bytecode level, for the sake of readability, the examples are also given on source level

```
final class Code_Table {
  private/*@spec_public */short tab[];

  //@invariant tab != null;

  ...

  //@requires size <= tab.length;
  //@ensures true;
  //@exsures (Exception) false;
  public void clear(int size) {
  1  int code;
  2  //@loop_modifies code, tab[*];
  3  //@loop_invariant code <= size && code >= 0;
  4  for (code = 0; code < size; code++) {
  5    tab[code] = 0;
     }
  }
}
```

Figure 4.2: A JML-annotated method

a JML specification. The method `clear` declared in class `Code_Table` receives an integer parameter `size` and assigns `0` to all the elements in the array field `tab` whose indexes are smaller than the value of the parameter `size`. The specification of the method guarantees that if every caller respects the method precondition and if every execution of the method guarantees its postcondition then the method `clear` never throws an exception of type or subtype `java.lang.Exception`[3]. This is expressed by the class and method specification contracts. First, a class invariant is declared which states that once an instance of type `Code_Table` is created, its array field `tab` is not null. The class invariant guarantees that no method will throw a `NullPointerException` when dereferencing (directly or indirectly) `tab`.

The method precondition requires the `size` parameter to be smaller than the length of `tab`. The normal postcondition, introduced by the keyword `ensures`, basically says that the method will always terminate normally, by declaring that the set of final states in case of normal termination includes all the possible final states, i.e. that the predicate `true` holds after the method's normal execution[4]. On the other hand, the exceptional postcondition for the exception `java.lang.Exception` says that the method will not throw any exception of type `java.lang.Exception` (which includes all runtime

[3]Note that every Java runtime exception is a subclass of `java.lang.Exception`

[4]Actually, after terminating execution the method guarantees that the first `size` elements of the array tab will be equal to 0, but as this information is not relevant to proving that the method will not throw runtime exceptions we omit it

exceptions). This is done by declaring that the set of final states in the exceptional termination case is empty, i.e. the predicate `false` holds if an exception caused the termination of the method. The loop invariant says that the array accesses are between index `0` and index `size - 1` of the array `tab`, which guarantees that no loop iteration will cause a `ArrayIndexOutOfBoundsException` since the precondition requires that `size <= tab.length`.

Once the source code is completed by the JML specification, the Java source is compiled using a normal non-optimizing Java compiler that generates debug information like LineNumberTable and LocalVariableTable, needed for compiling the JML annotations. From the resulting class file and the specified source file, the JML annotations are compiled into BCSL and inserted into user-defined attributes of the class file.

For generating the verification conditions, we use a bytecode verification condition generator (vcGen) based on a bytecode weakest precondition calculus [37].

From Program Proofs to Program Optimizations

In this phase, the bytecode instructions that can safely be executed without runtime checks are identified. Depending on the complexity of the verification conditions, Jack can discharge them to the fully automatic prover Simplify, or to the Coq and AtelierB interactive theorem prover assistants. There are several conditions to be met for a bytecode instruction to be optimized safely – the precondition of the method the instruction belongs to must hold every time the method is invoked, and the verification condition related to the exceptional termination must also hold. Once identified, proved instructions can be marked in user-defined attributes of the class file so that the compiler can find them.

More Precise Optimizations

As we discussed earlier, in order to optimize an instruction in a method body, the method precondition must be established at every call site and the method implementation must be proved not to throw an exception under the assumption that the method precondition holds. This means that if there is one call site where the method precondition is broken then no instruction in the method body will be optimized.

Actually, the analysis may be less conservative and therefore more precise. We illustrate with an example how one can achieve more precise results.

Consider the example of figure 4.3. On the left side of the figure, we show source code for method `setTo0` which sets the `buff` array element at index `k` to 0. On the right side, we show the bytecode of the same method. The `iastore` instruction at index `3` may throw two different runtime exceptions: `NullPointerException`, or `ArrayIndexOutOfBoundException`. For the method execution to be safe (i.e. no Runtime exception is thrown), the method requires some certain conditions to be fulfilled by its callers. Thus, the method's precondition states that the `buff` array parameter must not be null and that the `k` parameter

```
...

//@requires buff != null;
//@requires k >= 0 ;
//@requires k <= buff.length;
//@ensures true;
//@exsures (Exception) false;              0 aload_2
public void setTo0(int k,int[] buff)       1 iload_1
{                                          2 iconst_0
  buff[k] = 0;                             3 iastore
}                                          4 return
```

Figure 4.3: The source code and bytecode of a method that may throw several exceptions

must be inside the bounds of `buff`. If at all call sites we can establish that the `buff` parameter is always different from null, but there are sites at which an unsafe parameter `k` is passed the optimization for `NullPointerException` is still safe although the optimization for `ArrayIndexOutOfBoundException` is not possible. In order to obtain this kind of preciseness, a solution is to classify the preconditions of a method with respect to what kind of runtime exception they protect the code from. For our example, this classification consists of two groups of preconditions. The first is related to `NullPointerException`, i.e. `buff != null` and the second consists of preconditions related to `ArrayIndexOutOfBoundException`, i.e. `k >= 0 && k <= buff.length`. Thus, if the preconditions of one group are established at all call sites, the optimizations concerning the respective exception can be performed even if the preconditions concerning other exceptions are not satisfied.

4.5 Memory consumption

Another application is a framework to perform a precise analysis of resource consumption for Java bytecode programs (for the clarity of the explanations all examples in the introduction deal with source code). In order to illustrate the principles of our approach, let us consider the following program:

```
public void m (A a) {
  if (a == null)
    { a = new A();  }
  a.b = new B(); }
```

In order to model the memory consumption of this program, we introduce a *ghost* (or, *model*) variable `Mem` that accounts for memory consumption; more precisely, the value of `Mem` at any given program point is meant to provide an upper bound to the amount of

memory consumed so far. To keep track of the memory consumption, we perform immediately after every bytecode that allocates memory an increment of `Mem` by the amount of memory consumed by the allocation. Thus, if the programmer specifies that `ka` and `kb` is the memory consumed by the allocation of an instance of class `A` and `B` respectively, the program must be annotated as:

```
public void m (A a) {
 if (a == null)
   {a = new A(); //set Mem = Mem + ka;}
 a.b = new B(); //set Mem = Mem + kb; }
```

Such annotations allow to compute at run-time the memory consumed by the program. However, we are interested in static prediction of memory consumption, and resort to pre- and postconditions to this end.

Even for a simple example as above, one can express the specification at different levels of granularity. For example, fixing the amount of memory that the program may use, `Max`, one can specify that the method will use at most `ka` + `kb` memory units and will not overpass the authorized limit `Max`, with the following specification:

$$\begin{array}{l} \texttt{//@ requires} \quad \texttt{Mem} + \texttt{ka} + \texttt{kb} \leq \texttt{Max} \\ \texttt{//@ ensures} \quad \texttt{Mem} \leq \texttt{\old(()Mem)} + \texttt{ka} + \texttt{kb} \end{array}$$

```
public void m (A a) { ... }
```

Or try to be more precise and relate memory consumption to inputs with the following specification:

$$\begin{array}{l} \texttt{//@ requires a==null} \Rightarrow \texttt{Mem} + \texttt{ka} + \texttt{kb} \leq \texttt{Max} \\ \qquad\qquad\qquad\qquad \wedge\ \texttt{!(a==null)} \Rightarrow \texttt{Mem} + \texttt{kb} \leq \texttt{Max} \\ \texttt{//@ ensures} \\ \qquad \texttt{\old(a)==null} \Rightarrow \texttt{Mem} \leq \texttt{\old(()Mem)} + \texttt{ka} + \texttt{kb} \\ \qquad \wedge\ \texttt{!(\old(a)==null)} \Rightarrow \texttt{Mem} \leq \texttt{\old(()Mem)} + \texttt{kb} \end{array}$$

```
public void m (A a) { ... }
```

More complex specifications are also possible: one can take into account whether the program will throw an exception or not by using (possibly several) exceptional postconditions stating that $\texttt{k}_\texttt{E}$ memory units are allocated in case the method exits on exception `E`.

The main characteristics of our approach are:

- *Precision:* our analysis allows to specify and enforce precise memory consumption policies, including those that take into account the results of branching statements or the values of parameters in method calls. Being based on program logics, which are very versatile, the precision of our analysis can be further improved by using it in combination with other analysis, such as control flow analysis and exception analysis;

- *Correctness:* our analysis exploits existing program logics which are (usually) already known to be sound. In fact, it is immediate to derive the soundness of our analysis from the soundness of the program logic, provided ghost annotations that update memory consumption variables are consistent with an instrumented semantics that extends the language operational semantics with a suitable cost model that reflects resource usage;

- *Language coverage:* our analysis relies on the existence of a verification condition generator for the programming language at hand, and is therefore scalable to complex programming features. In the course of the paper, we shall illustrate applications of our approach to programs featuring recursive methods, method overriding and exceptions;

- *Usability:* our approach can be put to practice immediately using existing verification tools for program logics. We have applied it to annotated Java bytecode programs using a verification environment developed in [5]. It is also possible to use our approach on JML annotated Java source code [6], and more generally on programs that are written in a language for which appropriate support for contract-based reasoning exists;

- *Annotation and proof generation:* in contrast to other techniques discussed above, our approach requires user interaction, both for specifying the program and for proving that it meets its specification. In order to reduce the burden of the user, we have developed heuristics that infer automatically part of the annotations, and use automatic procedures to help discharge many proof obligations automatically.

Furthermore, our analysis may be used to guarantee that no memory allocation is performed in undesirable states of the application, namely after initialization or during a transaction in a Java Card.

On the negative side, our method does not deal with garbage collection nor arrays.

4.5.1 Modeling Memory Consumption

The objective of this section is to demonstrate how the user can annotate and verify programs in order to obtain an upper bound on memory consumption. We begin by describing the principles of our approach, then turn to establish its soundness, and finally show how it can be applied to non-trivial examples involving recursive methods and exceptions.

Principles

Let us begin with a very simple memory consumption policy which aims at enforcing that programs do not consume more than some fixed amount of memory `Max`. To enforce this policy, we first introduce a ghost variable `Mem` that represents at any given point of the

program the memory used so far. Then, we annotate the program both with the policy and with additional statements that will be used to check that the application respects the policy.

The precondition of the method `m` should ensure that there must be enough free memory for the method execution. Suppose that we know an upper bound of the allocations done by method `m` in any execution. We will denote this upper bound by `mthdCon(m )`. Thus there must be at least `mthdCon(m )` free memory units from the allowed `Max` when method `m` starts execution. Thus the precondition for `m` is:

$$//@\ \texttt{requires}\ \texttt{Mem} + \texttt{mthdCon(m)} \leq \texttt{Max}.$$

The precondition of the program entry point (i.e., the *main* method from which an application may start its execution) should also give the initial memory used by the virtual machine, i.e. require that variable `Mem` is equal to some fixed constant.

The normal postcondition of the method `m` must guarantee that the memory allocated during a normal execution of `m` is not more than some fixed number `mthdCon(m )` of memory units. Thus for the method `m` the postcondition is:

$$//@\ \texttt{ensures}\ \texttt{Mem} \leq \backslash\texttt{old(()Mem)} + \texttt{mthdCon(m)}.$$

The exceptional postcondition of the method `m` must specify that the memory allocated during an execution of `m` terminating by throwing an exception `Exception` is not more than `mthdCon(m )` units. Thus for the method `m` the exceptional postcondition is:

$$\begin{aligned} //@\ &\texttt{exsures(Exception)} \\ &\texttt{Mem} \leq \backslash\texttt{old(()Mem)} + \texttt{mthdCon(m)}. \end{aligned}$$

For every instruction that allocates memory the ghost variable `Mem` must be updated accordingly. For the purpose of this paper, we only consider dynamic object creation with the bytecode `new`; arrays are left for future work and briefly discussed in the conclusion.

In order to perform the update for `new` bytecodes, we assume given a function `allocInst` : $Class \rightarrow int$ gives an estimation of the memory used by an instance of a class. Then at every program point where a bytecode `new A` is found, the ghost variable `Mem` must be incremented by `allocInst(A)`. This is achieved by inserting a ghost assignment associated with any `new` instruction, as shown below:

$$\texttt{new A} \quad //\texttt{set Mem} = \texttt{Mem} + \texttt{allocInst(A)}.$$

Correctness

An important question is whether our approach guarantees that the memory allocated by a given program conforms to the memory consumption policy imposed by BCSL annotations. We can prove that our approach is correct by instrumenting the operational semantics of the bytecode language to reflect memory consumption. Concretely, this is achieved by extending states with the special variable `Mem`, and describing for each bytecode and for ghost assignments the effect of the weakest precondition calculus on `Mem` (in the fragment of the language considered, the only instruction to modify memory is `new`, thus the only instruction whose weakest precondition calculus has an effect on `Mem` is `new`).

We can then prove the correctness of the annotations w.r.t. the instrumented operational semantics, under the proviso that ghost assignments triggered by object creation are compatible with the instrumented operational semantics.

4.5.2 Inferring Memory Allocation

In the previous section, we have described how the memory consumption of a program can be modeled in BCSL and verified using an appropriate verification environment. While our examples illustrate the benefits of our approach, especially regarding the precision of the analysis, the applicability of our method is hampered by the cost of providing the annotations manually. In order to reduce the burden of manually annotating the program, one can rely on annotation assistants that infer automatically some of the program annotations (indeed such assistants already exist for loop invariants [35] and class invariants [32]). In this section, we describe an implementation of an annotation assistant dedicated to the analysis of memory consumption, and illustrate its functioning on an example.

The annotation assistant performs two tasks. First, it inserts the ghost assignments on appropriate places; for this task, the user must provide annotations about the memory required to create objects of the given classes.

Second, it inserts pre- and postconditions for each method. In this case, variants for loops and recursive methods may be given by the user or be synthesized through appropriate mechanisms. Based on this information, the annotation assistant recursively computes the memory allocated on each loop and method. Essentially, it finds the maximal memory that can be allocated in a method by exploring all its possible execution paths.

The function `mthdCon(.)` is defined as follows:

- **Input:** Annotated bytecode of a method `m` , and memory policies for methods that are called by `m` ;
- **Output:** Upper bound of the memory allocated by `m` ;
- **Body:** The first step is to compute the loop structure of the method, then to compute an upper bound to the memory allocated by each loop using its variant,

and then to compute an upper bound to the memory allocated along each execution path.

The annotation assistant currently synthesizes only simple memory policies (i.e., whenever the memory consumption policy does not depend on the input). Furthermore, it does not deal with arrays, subroutines, nor exceptions, and is restricted to loops with a unique entry point. The latter restriction is not critical because it accommodates code produced by non-optimizing compilers. However, a pre-analysis could give us all the entry points of more general loops, for instance by the algorithms given in [8]; our approach may be thus applied straightforwardly. How to treat arrays is briefly discussed in the conclusion.

Chapter 5

Conclusion

Bibliography

[1] Jean-Raymond Abrial. *The B Book, Assigning Programs to Meanings*. Cambridge University Press, 1996.

[2] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers-Principles, Techniques and Tools*. Addison-Wesley: Reading, 1986.

[3] B.Meyer. *Object-Oriented Software Construction*. Prentice Hall, 2 revised edition, 1997.

[4] Richard Bornat. Proving pointer programs in Hoare Logic. In *MPC*, pages 102–126, 2000.

[5] L. Burdy and M. Pavlova. Annotation carrying code. Manuscript, 2004.

[6] L. Burdy, A. Requet, and J.-L. Lanet. Java applet correctness: A developer-oriented approach. In K. Araki, S. Gnesi, and D. Mandrioli, editors, *FME 2003: Formal Methods: International Symposium of Formal Methods Europe*, volume 2805 of *Lecture Notes in Computer Science*, pages 422–439. Springer-Verlag, 2003.

[7] Lilian Burdy and Antoine Requet. Jack : Java Applet Correctness Kit. In *GDC 2002, Singapore*, November 2002.

[8] D. Cachera, T. Jensen, D. Pichardie, and G. Schneider. Certified memory usage analysis. In J. Fitzgerald, I. Hayes, and A. Tarlecki, editors, *Proceedings of FM'05*, volume 3582 of *Lecture Notes in Computer Science*, pages 91–106, 2005. To appear.

[9] Yoonsik Cheon. A runtime assertion checker for the Java Modeling Language. Technical Report 03-09, Department of Computer Science, Iowa State University, Ames, IA, April 2003. The author's Ph.D. dissertation. Available from `archives.cs.iastate.edu`.

[10] Yoonsik Cheon and Gary T. Leavens. A runtime assertion checker for the Java Modeling Language (JML). In Hamid R. Arabnia and Youngsong Mun, editors, *the International Conference on Software Engineering Research and Practice (SERP '02)*, pages 322–328. CSREA Press, June 2002.

[11] David L. Detlefs, K. Rustan M. Leino, Greg Nelson, and James B. Saxe. Extended static checking. Research Report 159, Compaq Systems Research Center, December 1998.

[12] Krishna Kishore Dhara and Gary T. Leavens. Forcing behavioral subtyping through specification inheritance. In *18th International Conference on Software Engineering*, pages 258–267. IEEE Computer Society Press, 1996.

[13] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2):1–25, 2001.

[14] 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.

[15] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. In *ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI'2002)*, pages 234–245, 2002.

[16] Cormac Flanagan and James B. Saxe. Avoiding exponential explosion: Generating compact verification conditions. In *Conference Record of the 28th Annual ACM Symposium on Principles of Programming Languages*, pages 193–205. ACM, January 2001.

[17] Lisa Friendly. The design of distributed hyperlinked programming documentation. In S. Fraïssè, F. Garzotto, T. Isakowitz, J. Nanard, and M. Nanard, editors, *IWHD'95*, pages 151–173. Springer, 1995.

[18] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison Wesley, 1996.

[19] G. Grimaud and J.-J. Vandewalle. Introducing research issues for next generation Java-based smart card platforms. In *Proc. Smart Objects Conference (sOc'2003)*, Grenoble, France, 2003.

[20] Marieke Huisman. *Java Program Verification in Higher-Order Logic with PVS and Isabelle*. PhD thesis, University of Nijmegen, The Netherlands, 2001.

[21] Kazuaki Ishizaki, Motohiro Kawahito, Toshiaki Yasue, Mikio Takeuchi, Takeshi Ogasawara, Toshio Suganuma, Tamiya Onodera, Hideaki Komatsu, and Toshio Nakatani. Design, implementation, and evaluation of optimizations in a just-in-time compiler. In *JAVA '99: Proceedings of the ACM 1999 conference on Java Grande*, pages 119–128, New York, NY, USA, 1999. ACM Press.

[22] B. Jacobs, J. Kiniry, and M. Warnier. Java program verification challenges. In F. S. de Boer, M. M. Bonsangue, S. Graf, and W.-P. de Roever, editors, *Formal Methods for Components and Objects, volume 2852*, lncs, pages 202–219. Springer, Berlin, 2003.

[23] Laurent Lagosanto. Next-generation embedded java operating system for smart cards. In *4th Gemplus Developer Conference*, 2002.

[24] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. JML: A notation for detailed design. In Haim Kilov, Bernhard Rumpe, and Ian Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, pages 175–188. Kluwer Academic Publishers, Boston, 1999.

[25] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Technical Report 98-06u, Iowa State University, Department of Computer Science, April 2003. See `www.jmlspecs.org`.

[26] Gary T. Leavens, Yoonsik Cheon, Curtis Clifton, Clyde Ruby, and David R. Cok. How the design of JML accommodates both runtime assertion checking and formal verification. Technical Report 03-04, Department of Computer Science, Iowa State University, Ames, Iowa, 50011, March 2003. To appear in the proceedings of FMCO 2002.

[27] G.T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. Cok, and J. Kiniry. *JML Reference Manual*.

[28] K. Rustan M. Leino. Extended static checking: A ten-year perspective. In Reinhard Wilhelm, editor, *Informatics—10 Years Back, 10 Years Ahead*, volume 2000 of *LNCS*. Springer, 2000.

[29] K. Rustan M. Leino, Greg Nelson, and James B. Saxe. ESC/Java user's manual. Technical Note 2000-002, Compaq SRC, October 2000.

[30] K. Rustan M. Leino, James B. Saxe, and Raymie Stata. Checking Java programs via guarded commands. Technical Note 1999-002, Compaq SRC, May 1999.

[31] Tim Lindholm and Frank Yellin. *Java Virtual Machine Specification*. Addison-Wesley Longman Publishing Co., Inc., 1999.

[32] F. Logozzo. Automatic inference of class invariants. In G. Levi and B. Steffen, editors, *Proceedings of VMCAI'04*, volume 2937 of *Lecture Notes in Computer Science*, pages 211–222. Springer-Verlag, 2004.

[33] Renaud Marlet and Daniel Le Metayer. Security properties and Java Card specificities to be studied in the SecSafe project. Technical Report SECSAFE-TL-006, Trusted Logic, August 2001. Available from `http://www.doc.ic.ac.uk/~siveroni/secsafe/docs.html`.

[34] Deepak Mulchandani. Java for embedded systems. *Internet Computing, IEEE*, 2(3):30 – 39, 1998.

[35] Jeremy W. Nimmer and Michael D. Ernst. Automatic generation of program specifications. In *ISSTA 2002, International Symposium on Software Testing and Analysis*, pages 232–242, Rome, Italy, 2002.

[36] 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, *Proceedings of CARDIS'04*. Kluwer, 2004.

[37] Mariela Pavlova. Java bytecode logic and specification. Technical report, INRIA, Sophia-Antipolis, 2005. Draft version.

[38] Erik Poll, Pieter Hartel, and Eduard de Jong. A Java reference model of transacted memory for smart cards. In *Smart Card Research and Advanced Application Conference (CARDIS'2002)*, pages 75–86. USENIX, 2002.

[39] Erik Poll, Joachim van den Berg, and Bart Jacobs. Formal specification of the Java Card API in JML: the APDU class. *Computer Networks*, 36(4):407–421, 2001.