Java Applet Correctness: a Developer-Oriented Approach

L. Burdy, A. Requet, and J.-L. Lanet

Gemplus Research Labs
La Vigie
Avenue du Jujubier - ZI Athelia IV
13705 La Ciotat CEDEX - France
{Lilian.Burdy,Antoine.Requet,Jean-Louis.Lanet}@gemplus.com

Abstract. This paper presents experiments on formal validation of Java applets. It describes a tool that has been developed at the Gemplus Research Labs. This tool allows to formally prove Java classes annotated with JML, an annotation language for Java that provides a framework for specifying class invariants and methods behaviours. The foundations and the main features of the tool are presented. The most innovative part of the tool is that it is tailored to be used by Java programmers, without any particular background in formal methods. To reduce the difficulty of using formal techniques, it aims to provide a user-friendly interface which hides to developers most of the formal features and provides a "Java style view" of lemmas.

Keywords: *Java, Correctness Proof, Proof User Interface*

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 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 us to increase the quality, but also to reduce validation costs.

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 [10].

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. Java developers should be able to validate their code, or at least to get a good assurance on its correctness.

This paper presents such a tool: the Java Applet Correctness Kit (or `JACK`). This tool, already briefly described in [2], is a formal tool that allows one to prove properties on Java programs using the Java Modeling Language [15] (JML). Its application domain is, at the moment, smart card applets, but one can consider that it can be useful in many development contexts. It generates proof obligations allowing to prove that the Java code conforms to its JML specification. The lemmas are translated into the B language [1], allowing to use the automatic prover developed within the B method.

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 convience on code correctness.

The remainder of the paper is organized as follow. Section 2 describes JML and the different tools supporting it. Section 3 presents the architecture and the main principles of the tool we have developed. Section 4 describes more precisely the innovative parts of the tool and explains why we consider it as accessible to any developers. Section 5 describes experiments on an applet and the metrics that have been collected. Section 6 presents research perspectives and Section 7 concludes.

2 Java Modeling Language

This section briefly presents JML and the tools that have been developed around it. JML [15] is a language that allows one to specify Java classes by formally expressing properties and requirements on those classes and their methods. Some keywords and logical constructions have been added to Java, but the core expression language is close to Java. JML benefits from Java's precise semantics. JML has also been defined so that specifications are easy to read and write by Java programmers. Taking those facts into account, many tools have been developed around JML annotations.

2.1 Specifying Java applets

Figure 1 presents an example JML specification. The language provides keywords to specify:

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

- *Class invariants*: invariants correspond to properties on member and class variables that must always hold (from an external observer point of view, since invariants are not required to hold inside method implementations), and are introduced using the `invariant` keyword. In the example, the integer `i` is positive.
- *Preconditions*: preconditions are associated to methods, and correspond to properties that must hold in order to call the method. The `requires` keyword is used to define preconditions. In the example, the method parameter `c` is required to be non-null.
- *Postconditions*: as preconditions express the properties that must be true when calling a method, postconditions describes the behavior of the method by expressing the properties ensured by the method. They are expressed using the `ensures` keyword. In the example, the value of `i` will become 3 for the current instance and 2 for the parameter `c`. Special postconditions can also be used to describe exceptional behaviors, in particular, when the method throws an exception. JML uses the special `exsures` keyword to define those special postconditions. In the example, it is expressed that the method will not throw any exceptions.

Additionally, JML requires specifying which variables can be modified by a method. This is specified with the `modifies` keyword. In the example, it is specified that only the variables `i` will be modified for the instances `this` and `c` by the method `m`.

```
class C {
   short i;
   //@ invariant i >= 0;

   //@ requires c != null;
   //@ modifies i, c.i;
   //@ ensures i == 3 && c.i == 2;
   //@ exsures (Exception) false;
   void m(C c) {
      i = 3;
      c.i = 2;
   }
}
```

Fig. 1. JML example

The language contains more complex constructions that allows one to model more complex behaviors. Some realistic examples have already been modeled using JML: the Java Card API [18], part of the Java Standard Edition API [13], or a banking applet for smart cards [5]. Those examples show that using

JML is a realistic way to model Java programs, especially with the Java Card[2] restrictions.

2.2 Tools

Before presenting the tool that we have developed, we describe, in this section, the other existing tools that are supporting JML in order to compare them with our approach. An overview on tools supporting JML is presented in [16]. From this time, different new tools have been developed and existing tools have been improved. Three categories can be distinguished: runtime checkers, static validation tools and proof tools.

Runtime checker A runtime checker is part of the JML release. The JML release consists of different tools:

– a type-checker, allowing to verify the syntax of the JML specifications,
– the `jmldoc` tool, that is similar to `JavaDoc`, but adds the JML specification to the generated html documentation and
– the `jmlc` tool [6], that uses the JML annotation in order to add runtime assertions in the generated code.

The assertion checking allows running the code with dynamic tests checking for the correctness of the preconditions, the postconditions and the invariants. Thus, problems can be found early, as a specification violation will generate false assertions, potentially before introducing a visible runtime error. `jmlc` is integrated with `Junit`[3] giving `jmljunit` [7]. This tool generates an oracle and skeletons used by `Junit` to run test cases.

Static validation tools The main tool in this category is the `ESC/Java` [9] static checker for Java. It performs a static analysis of a Java source file in order to check for potential errors in the program. This tool does not aim to provide a formal assurance that the verified class is correct. The spirit of `ESC/Java` is to be a lightweight tool that aims to be used during development in order to identify and correct bugs early. For example, it is really efficient to warn about potential null pointer usage, and provides counter-examples when a property expressed could be erroneous. An experiment on `ESC/Java` is notably presented in [5].

Proof tools This is the category that our tool, `JACK`, belongs to. The idea behind those tools is to convert the JML annotated source code into formal models. Such a conversion allows to reason mathematically on the program, and

[2] Java Card is a standard defined by Sun Microsystems tailored to smart card. It is a subset of Java. It does not support threads, multi-dimensional arrays, floating point types, etc.
[3] JUnit is a regression testing framework, see `http://www.junit.org`

to achieve correctness proofs. Those proof tools are targeted to Java Card, which does not contain complex Java features which would be difficult to handle such as, for example, multi-threading.

- The `LOOP` tool [14, 12, 19] is a tool converting Java annotated sources to PVS models. It treats the complete Java Card language and now proposes an automated proof obligation generation using weakest precondition calculus.
- The Java Interactive Verification Environment (`Jive`) [17] aims to also translate JML annotated Java in PVS models. It proposes an interactive environment to deal with the proof obligation generation. Nevertheless, it does not handle all the Java Card language.
- The `Krakatoa` tool [8] is the more recent one. It aims to translate Java annotated sources into an internal language from which proof obligations are generated into Coq[4].

All those tools are actually developed within the VerifiCard[5] project. They all have the same goal but approaches are slightly different.

3 Foundations

The previous section presented several tools that exist to handle JML annotated Java programs. However, we feel that the aspects of user friendliness and automatization are not sufficiently addressed by these tools, therefore we decided to develop our own tool.

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 (in contrast to for example `LOOP`). This interface is described section 4;
- 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. This is in contrast with the `LOOP` and `Jive` approach, where users need in-depth PVS knowledge to be able to do verifications;
- 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 (in contrast to `ESC/Java`). Nevertheless the tool is not formally developed, as `LOOP` is. 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 relatively independent of the particular prover used, so that if the use of another prover is required (for example by a certification institute) it is relatively easy to adapt the tool accordingly.

[4] Coq is a proof assistant developed by the INRIA: `http://coq.inria.fr`
[5] European IST Project: `http://www.verificard.org`

This section presents the tool architecture and its principles, concerning object-oriented concepts formalization and lemma generation.

3.1 Architecture

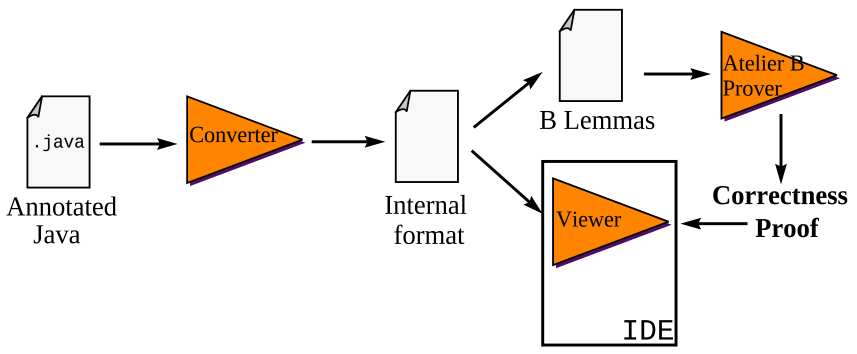


Fig. 2. `JACK` architecture

Figure 2 presents an overview of the `JACK` architecture. `JACK` consists of two parts: a converter (a lemma generator) from Java source annotated with JML into B 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 4.2. This part focuses on the converter.

The `JACK` converter converts a Java class into a B model and allows to prove properties. The B method [1] is a formal method with different features such as refinement schema, translation in C code, etc. We only use it as a framework that allows us to express lemmas and to prove them using a prover. This prover is part of the `Atelier B`[6] tool and provides two modules: an automatic one and an interactive one. The choice of the B method as the targeted prover is due to different reasons. The `Atelier B` prover is quite well automated with proof rate approaching 80%. It can also easily be customized to increase this rate by adding new proof tactics and theorems. Nevertheless, dealing with the interactive interface which allows to prove remaining lemmas is still an activity reserved to experts. Another more pragmatic reason is the good experience of our team in the `Atelier B` prover (see [3, 4]).

Our goal is to prove properties on source files written with the Java language. To reach this goal, one has to know how to “translate” a Java source file in B lemmas. The two main issues are:

[6] `Atelier B` is a CASE tool developed by Clearsy (`http://www.clearsy.com`) that allows one to develop using the B method.

1. How to formalize the object-oriented Java features in set theory (the B language is a first order language with set theory)?
2. How to generate lemmas from Java methods?

The following sections provide answers to those questions.

3.2 Object-oriented concepts formalization

The adopted solution concerning the formalization of object-oriented concept is to generate lemmas with the point of view of one Java class. Each generated lemma for a class contains, as prelude, the formalization of the memory model instantiated with the context of the current class, i.e. the class hierarchy that it can access, all the fields and methods that it can use. The prelude contains two parts: a generic one and a contextual one.

$$\begin{array}{ll}
\text{SET} & \text{REFERENCES} \\
\text{null} & \in \text{REFERENCES} \\
\text{subtypes} & \in \text{TYPES} \leftrightarrow \text{TYPES} \\
\text{instances} & \subset \text{REFERENCES} \\
\text{null} & \notin \text{instances} \\
\text{typeof} & \in \text{instances} \rightarrow \text{TYPES}
\end{array}$$

Fig. 3. Memory model representation

The generic prelude contains many definitions concerning built-in types with their operators. It contains also the memory model representation (see Figure 3). This model is defined by an infinite set of possible references, a particular reference corresponding to null, a subtyping relation, the set of currently allocated instances and a function associating a type to each instance. This model is completed with definitions allowing to handle arrays that are not given here. This set of constants and variables represents the memory state associated to each lemma, corresponding, for instance, to the state of the memory at the beginning of an operation. In this model, an object creation, for instance, will be defined as taking an arbitrary element from the set of references (different from the special constant null), adding this reference to the set of instances and assigning a type to it.

The set of types is defined by $\text{TYPES} = \text{NAMES} * \mathbb{N}$, where NAMES corresponds to the set of classes referenced by the program. This definition allows to handle the types corresponding to array types: a type corresponds to a class and a dimension represented by a natural number (for objects, this number is always 0). For instance the Java type Object will correspond to the type $\text{c\_Object} \mapsto 0$ and the Java type Object[][][] to $\text{c\_Object} \mapsto 3$.

The set NAMES is not generic since it depends on the reachable classes from the current one. It does not belong to the common prelude but to the contextual one. For instance, the valuation of NAMES in the case of the example in Figure 1 (where classes names are prefixed to avoid names conflict) is

$$\mathrm{NAMES} = \left\{ \begin{array}{l} \mathrm{c\_int}, \mathrm{c\_short}, \mathrm{c\_char}, \mathrm{c\_byte}, \mathrm{c\_boolean}, \mathrm{c\_Object}, \\ \mathrm{c\_RuntimeException}, \mathrm{c\_Exception}, \mathrm{c\_Throwable}, \\ \mathrm{c\_NullPointerException}, \mathrm{c\_ArithmeticException}, \\ \mathrm{c\_ArrayIndexOutOfBoundsException}, \\ \mathrm{c\_NegativeArraySizeException}, \mathrm{c\_ClassCastException}, \\ \mathrm{c\_ArrayStoreException}, \mathrm{c\_C} \end{array} \right\}$$

One can notice that primitive types belongs to NAMES but they are only used to type arrays. For instance the Java type `short[]` will correspond to $\mathrm{c\_short} \mapsto 1$, but the type `short` will correspond to a prelude-defined type $\mathrm{t\_short} = -32768..32767$.

The subtyping relation is also valuated depending on the class hierarchy. It assigns to a class itself and all its subclasses and to an interface itself and all its implementing classes and interfaces. This gives, for example,

$$\begin{array}{l} \mathrm{subtypes}[\{\mathrm{c\_RuntimeException} \mapsto 0\}] \\ = \left\{ \begin{array}{l} \mathrm{c\_RuntimeException} \mapsto 0, \\ \mathrm{c\_NullPointerException} \mapsto 0, \\ \mathrm{c\_ArithmeticException} \mapsto 0, \\ \mathrm{c\_ArrayIndexOutOfBoundsException} \mapsto 0, \\ \mathrm{c\_NegativeArraySizeException} \mapsto 0, \\ \mathrm{c\_ClassCastException} \mapsto 0, \\ \mathrm{c\_ArrayStoreException} \mapsto 0 \end{array} \right\}. \end{array}$$

The fields are declared as variables, static fields are directly typed with their translated type, member fields are declared as functions from the set of instances of a type to the type of the field. For instance, the member field i (see Figure 1) is declared as follow:

$$f\_i \in \mathrm{typeof}^{-1}[\mathrm{subtypes}[\{\mathrm{c\_C} \mapsto 0\}]] \to \mathrm{t\_short}.$$

The invariants are declared as properties quantified over the instances. For instance, the invariant of the class C (see Figure 1) is declared as follow:

$$\forall c.(c \in \mathrm{instances} \wedge \mathrm{typeof(c)} \in \mathrm{subtypes}[\{\mathrm{c\_C} \mapsto 0\}] \Rightarrow 0 \leq \mathrm{f\_i(c)}).$$

The drawback of this approach is that one can not directly prove properties on the correctness of the formalization. But, proofs remain simpler as they remain centered on the specific case of a dedicated class.

3.3 Lemma generation

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 tried to implement a Weakest Precondition (WP) calculus to automate lemma generation. Huisman, in [11], 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` and `loop_exsures`. 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 `loop_exsures` allows us to specify the exceptional behavior of a loop. It is not necessary to apply the WP calculus but it can improve the understandability of the specification.

The two main drawbacks of the WP calculus are the loss of information and potential exponentional 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.

Exponentional 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 exponentional explosion usually occurs in a method with many sequenced branched statement (`if, switch`, etc.) Such methods usually perform different distinct sequenced treatments. Figure 4 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.

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.

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

Fig. 4. Specified block

4 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 developement environment is probably the main strength of `JACK`, as is the fact that the underlying mathematical formalism is not exposed to the user.

4.1 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 4.3).

Several other minor features are available to integrate within the development cycle, for example, reports on the status of the project can be generated as Microsoft Excel files.

4.2 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 5) of the lemma. The viewer displays

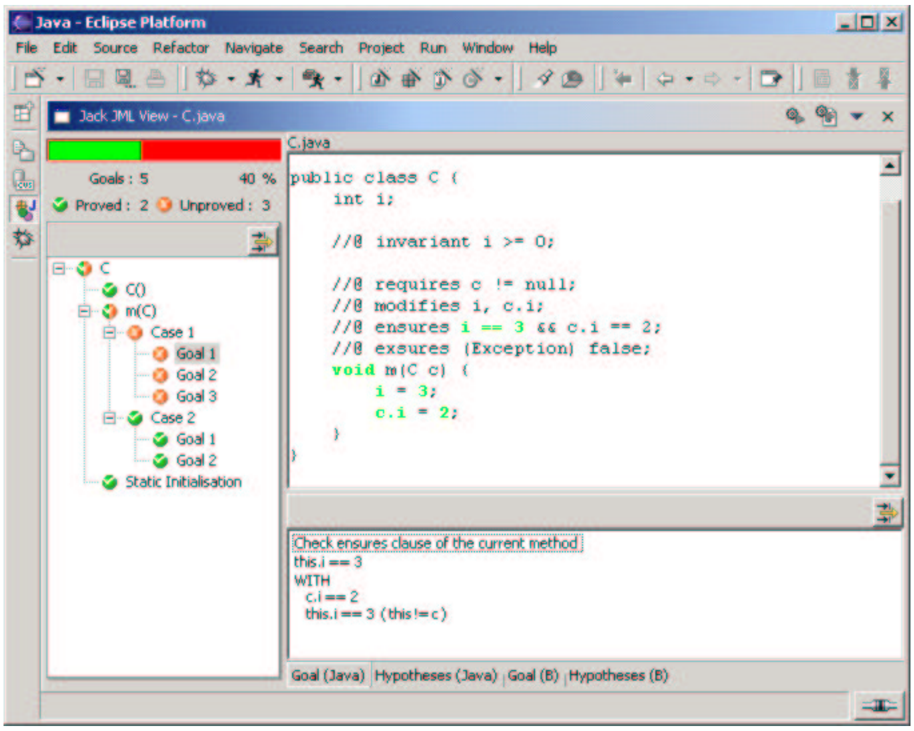
- information concerning the current proof status;
- the class methods with their lemmas;
- the source code;
- and the currently selected lemma (goals and hypotheses with Java or B presentation).

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.



```
public class C {
    int i;

    //@ invariant i >= 0;

    //@ requires c != null;
    //@ modifies i, c.i;
    //@ ensures i == 3 && c.i == 2;
    //@ exsures (Exception) false;
    void m(C c) {
        i = 3;
        c.i = 2;
    }
}
```

```
Check ensures clause of the current method
this.i == 3
WITH
  c.i == 2
  this.i == 3 (this != c)
```

Fig. 5. Viewer integrated in eclipse

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.

Java presentation of lemmas The hypothesis and goals of the current lemma are also displayed. As the conversion mechanism to B 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. To emphasize this point, we present some lemmas that have been generated from the method of Figure 1 concerning the post-condition correction. The ”B style view“ (Figure 6) presents two lemmas concerning the fact that `i` becomes 3 for the instance `this` and becomes 2 for the instance `c`. The fact that `i` has changed during the method is denoted using the overriding operator ($\mathbin{\lhd\!\!\!-}$ in the B language) twice, replacing the value of `i` for the instances `c` and `this`.

$$(i \mathbin{\lhd\!\!\!-} \{this \mapsto 3\} \mathbin{\lhd\!\!\!-} \{c \mapsto 2\})(this) = 3$$
$$(i \mathbin{\lhd\!\!\!-} \{this \mapsto 3\} \mathbin{\lhd\!\!\!-} \{c \mapsto 2\})(c) = 2$$

Fig. 6. B style view

The "Java style view“ (Figure 7) presents the same two lemmas. However, as Java has no operator equivalent to the B overriding operator, a special notation using the “`WITH`” keyword is used. This view allows to understand quickly which part of the specification is concerned by the lemma: the postcondition is presented as it appears in the code. Modifications concerning `i` are presented in a second part (under the `WITH` keyword) in a specific order, and possibly under constraints. In this example, the constraints express the fact that the value of `this.i` is equal to `3` under the assumption that `this` is different from `c`. This allows to understand rapidly that there is an error in the code in Figure 1, as it does not implement the requirements when `c` equals `this`. This can also be seen in the "B style view“ but it requires more expertise.

this.i == 3	c.i == 2
WITH	WITH
c.i == 2	c.i == 2
this.i == 3 (this != c)	this.i == 3 (this != c)

Fig. 7. Java style view

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 yet, and for which a Java notation is hard to define. For instance the set operators cannot be translated in a generic way. Handling such constructs will be performed by modifying the internal representation in order to better match the JML language. For instance the set $\in$ operator can be difficult to translate back into Java. However, as it is mainly used for representing the Java type information, adding a dedicated operator (that would be handled like a *in* operator from a B point of view) for representing this type information would allow an easy translation.

Such changes to the internal representation would also be useful for generating lemmas to other proof language such as Coq or PVS, for which a translation using the set operators may not be the most appropriate translation.

4.3 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.

However, it is expected that developers will be able to handle more and more of the interactive proofs. To achieve this, we provide limited support for interacting with the prover in order to allow users to prove common cases.

Currently, two interactions are provided:

– Identifying false hypothesis. In that case, the prover tries to prove the negation of the hypothesis, and if the proof succeeds, the goal is discharged.
– Showing "wrong pathes" in the source code. This is performed by clicking on a condition in the source code that "cannot happen". For instance, a condition within a `if` that is in contradiction with the preconditions.
  In that case, the corresponding hypothesis is retrieved and sent to the prover as a false hypothesis in a way similar to the previous case.

Those actions are still limited to one proof command that can either succeed or fail, and so cannot be used to perform full interactive proof yet. However, they still allow to discharge some classes of common proof obligations.

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.

5 Case study

To test JACK, we have developed a little banking application. This section presents different metrics concerning the evaluation of the tool on this package. Different

Classes	Java lines	JavaDoc lines	JML lines	Proof obligations	Automatic proof	Time to PO generate (s)	Time to prove (s)
Transfert_src	116	34	150	359	91%	22,5	238
AccountMan_src	105	51	236	269	82%	12,7	195
Currency_src	93	20	28	50	96%	7,6	17
Balance_src	64	38	58	335	95%	16,5	191
Spending_rule	40	33	62	42	67%	13,6	217

Table 1. banking applet metrics

remarks can be made from Table 1, concerning the cost of adding JML annotations, the performance of the tool, as well as the cost associated to the proof.

Cost of the annotations A first remark concerns the cost of the annotations. The metrics given here only concern the number of lines but one can see that the documentation size (JavaDoc and JML) is one and half greater than the code size. So, writing the JML specification seems to be a costly activity. This remark can be moderated by two points: this development was the first that we made, and annotations were added to already existing code. So it suffers from its lack of abstraction, and the annotations are really verbose. Moreover the time to specify is to be compared to the time to develop test data.

Responsiveness The automatic phases are quite responsive with some seconds to generate proof obligations. The automatic proof is not as cheap as the proof obligation generation and takes a few minutes. However, this still remains acceptable, since it is a non-blocking task that does not require users to wait for its completion.

For larger applets, it is however expected that the time required for the automatic proof will significantly increase.

Proof The automatic proof rating is much good. It is quite greater than the usual value for a B development (around 80%). This is mainly due to the fact that we used this applet as a test applet for extending the prover. So, as a side effect, the automatic prover is customized for this special applet.

Nevertheless, after automatic proof step, there remain 111 lemmas to prove using the Atelier B interface. An expert needs between 4 and 5 days to prove them.

6 Perspectives

At the moment, we have developed a prototype that is becoming a usable tool. We are beginning experimentation outside the lab. But we are considering that there are still many points where the approach can be improved.

6.1 Interactive proof cost

Currently, the interactive proof support is rather limited. Thus, proving the remaining proof obligations requires users to directly use the `Atelier B` tool interactive prover. Such a task can be tedious, since the B representation of the generated proof obligations can be hard to follow. Different perspectives exist to reduce the interactive proof cost.

Java interface Although full interactive proof will still be reserved to experts, providing an interface to the `Atelier B` interactive prover allowing to perform proofs by directly using the Java syntax would greatly improve the productivity of those experts. A first step would be to extend the Java view used to display the lemmas.

Test Activity Another way to reduce interactive proof activity is to balance it with testing activity. Methods where lemmas are not automatically proved could be tested. Using `JmlJunit` can be a good way to generate test skeleton on certain cases. A perspective is to integrate `JmlJunit` in our environment in manner to propose different validation level (full proof, proof mixed with test, etc).

Counter-example Another idea to reduce the cost of false lemma detection is to provide counter-example when it is possible. `ESC/Java` already tempts to give such counter-examples. Studies on that subject could give results helpful for developers to understand errors on the code or on the specifications.

6.2 Allowing expressions in the target formal language

Currently, `JACK` can be seen as a proof-obligations generator from JML annotated Java programs to B lemmas. A possible extension would be to add different target languages for lemma generation, for instance Coq or PVS. Such extension would allow using different provers for different lemmas. On the other side, it is also possible to enrich the JML specification by adding inline expressions or variables in a target language of the lemma generator. Such an extension would have a role similar to Java "native methods" at a specification level. That is, allowing to describe in a lower-level language things that cannot be described in JML (or that cannot be described efficiently from a proof point of view).

We are currently investigating such extension mechanisms, that would allow adding different languages for such use without having to modify the weakest-precondition calculus core.

7 Conclusion

The presented work allows formal methods experts to prove Java applet correctness. Moreover, it allow Java programmers to obtain an high assurance on their code correctness. This leads to the most important point: it allows non-experts to venture into the formal world. This is a necessary starting point for such validation techniques to be widely used.

The tool has been developed following a main objective: let Java developers validate their own code. We claim that JML is well suited to express low level design and conception choices and that usage of `JACK` can replace effectively unitary tests, giving developers the means to furnish code with good quality and non ambiguous documentation.

Taking benefits from recent research on Java program validation, we have developed an automated tool that generates lemmas from Java classes annotated with JML. The `JACK` principles are not really different from the `LOOP` team choices. Nevertheless, one can highlight some important differences. The `LOOP` tool describes formally Java semantics and the WP-calculus rules are applied by the prover. The main advantage is that the rules have been proven sound with regards to the semantics in the theorem prover. Our point of view, is more pragmatic: lemma are generated automatically, using Java developed rules that can only be checked by usual validation: test, code-inspection. The soundness of the tool cannot be formally proved but on the other hand a big effort is done to present lemmas to users in a way that he can understand them and verify that they are valid.

The tool is fully integrated in the eclipse IDE and presents lemmas in a visual way that allows developers to form their opinion on the lemma validity. An automatic prover discharges an important part of the lemmas. The remaining lemmas have to be proved using the prover interface. Often, this task cannot be done by developers. Different ways are studied to bypass it: expert support, test case generation, counter-example detection, etc.

We are now experimenting on real industrial products. We are trying to collect metrics in order to prove that this kind of validation is cost-saving, especially when the cost of testing is taken into account.

Acknowledgements
The authors would like to thank Marieke Huisman for its helpful comment on this paper.

References

1. Jean-Raymond Abrial. *The B Book, Assigning Programs to Meanings.* Cambridge University Press, 1996.
2. Lilian Burdy and Antoine Requet. Jack : Java Applet Correctness Kit. In *GDC 2002, Singapore*, November 2002.
3. Ludovic Casset. Development of an embedded verifier for Java Card byte code using formal methods. In Lars-Henrik Eriksson and Peter Alexander Lindsay, editors, *Formal Methods – Getting IT Right*, volume 2391 of *Lecture Notes in Computer Science*, pages 290–309. Springer-Verlag, July 22-24 2002.
4. Ludovic Casset, Lilian Burdy, and Antoine Requet. Formal development of an embedded verifier for Java Card byte code. In *DSN 2002, International Conference on Dependable Systems & Networks*, pages 51–56, Washington, D.C., USA, June 2002.
5. Néstor Cataño and Marieke Huisman. Formal specification and static checking of Gemplus' electronic purse using ESC/Java. In Lars-Henrik Eriksson and Peter Alexander Lindsay, editors, *Formal Methods – Getting IT Right*, volume 2391 of *Lecture Notes in Computer Science*, pages 272–289. Springer-Verlag, July 22-24 2002.
6. Yoonsik Cheon and Gary T. Leavens. A runtime assertion checker for the Java Modeling Language (JML). Technical Report 02–05, Department of Computer Science, Iowa State University, March 2002. In SERP 2002, pp. 322–328.
7. Yoonsik Cheon and Gary T. Leavens. A simple and practical approach to unit testing: The JML and JUnit way. In Boris Magnusson, editor, *ECOOP 2002 — Object-Oriented Programming, 16th European Conference, Máalaga, Spain, Proceedings*, volume 2374 of *Lecture Notes in Computer Science*, pages 231–255, Berlin, June 2002. Springer-Verlag.
8. Jean-Christophe Filliâtre, Claude Marché, Christine Paulin, and Xavier Urbain. Modeling of Java programs in Coq. in VeriSafe Workshop, Sept 2002.
9. Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for java. In Cindy Norris and James B. Fenwick Jr., editors, *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI-02)*, volume 37, 5 of *ACM SIGPLAN Notices*, pages 234–245, New York, June 17–19 2002. ACM Press.
10. James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification.* Addison Wesley, 1996.
11. Marieke Huisman. *Java Program Verification in Higher-Order Logic with PVS and Isabelle.* PhD thesis, University of Nijmegen, The Netherlands, 2001.
12. Marieke Huisman and Bart Jacobs. Java program verification via a Hoare logic with abrupt termination. In T. Maibaum, editor, *Fundamental Approaches to Software Engineering (FASE)*, volume 1783 of *Lecture Notes in Computer Science*, pages 284–303. Springer-Verlag, 2000.
13. Marieke Huisman, Bart Jacobs, and Joachim van den Berg. A case study in class library verification: Java's vector class. CSI Report CSI-R0007, Computing Science

Department, Nijmegen, March 2000.
`http://www.cs.kun.nl/csi//reports/full/CSI-R0007.ps.Z`.

14. Bart Jacobs and Erik Poll. A logic for the Java modeling language JML. In H. Hussmann, editor, *Fundamental Approaches to Software Engineering (FASE)*, volume 2029 of *Lecture Notes in Computer Science*, pages 284–299. Springer-Verlag, 2001.
15. Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Technical Report 98–06i, Iowa State University, Department of Computer Science, February 2000. `ftp://ftp.cs.iastate.edu/pub/techreports/TR98-06/TR.ps.gz`.
16. Gary T. Leavens, K. Rustan M. Leino, Erik Poll, Clyde Ruby, and Bart Jacobs. JML: notations and tools supporting detailed design in Java. In *OOPSLA 2000 Companion, Minneapolis, Minnesota*, pages 105–106. ACM, October 2000.
17. Jörg Meyer and Arnd Poetzsch-Heffter. An architecture for interactive program provers. In S. Graf and M. Schwartzbach, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 1785 of *Lecture Notes in Computer Science*, pages 63–77. Springer-Verlag, 2000.
18. Erik Poll, Joachim van den Berg, and Bart Jacobs. Specification of the JavaCard API in JML. In *Fourth Smart Card Research and Advanced Application Conference (IFIP Cardis)*. Kluwer Academic Publishers, 2000.
19. Joachim van den Berg and Bart Jacobs. The LOOP compiler for Java and JML. In T. Margaria and W. Yi, editors, *International Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 2031 of *Lecture Notes in Computer Science*, pages 299–312. Springer-Verlag, 2001.