

Specification and Runtime Verification of Java Card Programs

Umberto Souza da Costa¹, Anamaria Martins Moreira¹
Martin A. Musicante¹, Plácido A. Souza Neto²

¹DIMAp - Universidade Federal do Rio Grande do Norte
Campus Universitário – Lagoa Nova – Natal – RN – Brazil

²Centro Federal de Educação Tecnológica do Rio Grande do Norte
Caixa Postal 1559 – 59.015-000 – Natal – RN - Brazil

{umberto,anamaria,mam}@dimap.ufrn.br, placidoneto@cefetrn.br

***Abstract.** Java Card is a version of Java developed to run on devices with severe storage and processing restrictions. The applets that run on these devices are frequently intended for use in critical, highly distributed, mobile conditions. They are required to be portable and safe. Often, the requirements of the application impose the use of dynamic, on-card verifications, but most of the research developed to improve safety of Java Card applets concentrates on static verification methods. This work presents a runtime verification approach based on Design by Contract to improve the safety of Java Card applications. To this end, we propose JCML (Java Card Modeling Language) a specification language derived from JML (Java Modeling Language) and its implementation: a compiler that generates runtime verification code. We also present some experiments and quality indicators.*

1. Introduction

The Java Card programming language [Ortiz 2005] is a version of Java. Its programs are intended to run on very restricted architectures such as Smart Cards, SIM cards or security tokens. Many features and constructors of Java are not present in Java Card. These include some primitive types (such as `integer` or `float`) and most library classes. A specific version of the Java Virtual Machine (JVM) has been devised to run Java Card applets [Chen 2000]. The Java Card virtual machine includes support for atomic transactions, transient and persistent memory, as well as a firewall mechanism.

Java Card applets [Chen 2000] are usually deployed in highly distributed and mobile situations and tend to be developed for critical applications. The verification of such applets is often required to guarantee the intended behavior of the system to which these applets belong, in order to reduce financial and/or human risks.

The relevance of formal specification and verification methods for Java Card applications is reflected in the large number of works on this area. A typical example may be found in the Mondex case study [Jones and Woodcock 2008], developed in the context of the Grand Challenge on Verified Software, where the banks want to be sure that no money may be created in a system of electronic wallets.

The Java Modeling Language (JML) [Leavens and Cheon 2006, Leavens 2007] is a language designed to specify Java programs in detail. Software developers can use JML

to add specifications in accordance with the Design by Contract [Meyer 2000] principles by means of assertions, such as method preconditions and postconditions and class invariants. JML annotations can be automatically translated into runtime assertion checking code by JMLc [Cheon 2003, Bhorkar 2000], the JML compiler. JMLc produces Java executable bytecode supposed to run on any Java virtual machine where the JML runtime classes are available.

The usefulness of Design by Contract in general and JML for Java is already well established, as presented in [Leavens 2007]. Due to the critical nature of smartcard applications, runtime verifications associated to Design by Contract could contribute to the development of more robust code, e.g., by dealing with exceptional behaviour. However, JML and JMLc are *not* supported by the Java Card virtual machine. The input programming language accepted by the Java Card virtual machine has been restricted to cope with the restrictions imposed by the target devices where most of Java Card applets run. A consequence of this restriction is that Java Card cannot benefit from JML specification and verification tools in order to improve safety of its applets at runtime.

The motivation for our work is that, although the Java Card virtual machine is not able to deal with the code produced from a full JML specification, the safety of Java Card applets can be improved at least by a subset of JML. Such a subset can be defined in order to avoid all those features that are not supported by the Java Card virtual machine. It is necessary to ensure that both data and control structures involved in the specifications as well as the code generated for the verification of these specifications are compliant with the Java Card virtual machine.

In this context the main contributions of this paper are:

- The proposal of JCML, a restricted version of JML as a Java Card specification language.
- The design and implementation of a compiler for this language. Unlike the original JML compiler, our implementation focuses on the generation of concise and efficient code. Some simple optimization techniques are used for the generation of the Java Card code that will be run on-card.
- A case study where the size of the JCML-generated programs is compared to those programs generated by the standard JML compiler. The size of the generated code is the main restriction when considering constrained devices after compatibility with the Virtual Machine.

This paper is organized as follows: Section 2 presents Design by Contract and JML. Section 3 introduces the Java Card platform and discusses related limitations and advantages. After that, Section 4 shows how Java Card restrictions affect the choice of the JML subset compliant with the Java Card virtual machine, called JCML. Section 5 introduces the implementation of the JCML compiler, including the translation rules from JCML assertions to Java Card code. Experimental results are shown in section 6, where we compare the code produced by the JCML compiler and that produced by the JML compiler in terms of size. Finally, sections 7 and 8 present some related work and the next steps to be taken to improve JCML and its compiler, as well as our final remarks.

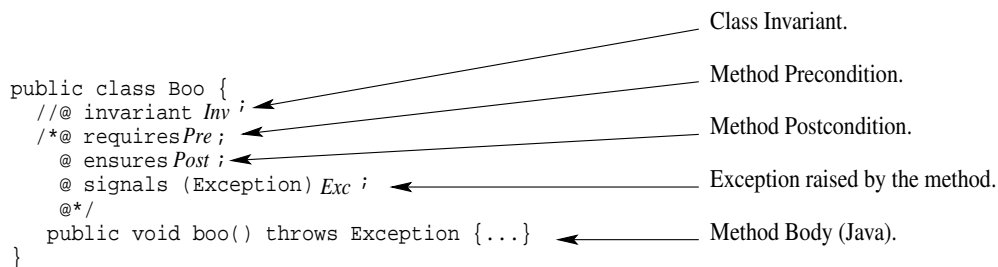
2. Design by contract and JML

Design by Contract [Meyer 1992, Meyer 2000] is a software development method based on the definition of contracts between software units. The method proposes the run-time verification of these contracts. Design by Contract uses logical assertions (preconditions, postconditions and class invariants) to specify contracts. These assertions can be added to the source code of a program and can be dynamically verified.

JML implements these ideas by using model-based specifications with mathematical concepts such as sets and first order logic to specify the contents of each contract condition. However, differently from other model-based specification languages such as Z [Woodcock and Davies 1996] and B [Abrial 1996], JML uses Java syntax as much as possible in its specifications, adding extra constructs to cope with specification concepts that are not directly expressed in Java, such as first order logic quantifiers.

In JML, assertions are expressed within the Java program by using a special kind of comments which express the conditions as first-order logic formulae, as follows:

```
public class Boo {  
  //@ invariant Inv ;  
  /*@ requires Pre ;  
  @ ensures Post ;  
  @ signals (Exception) Exc ;  
  @*/  
  public void boo() throws Exception {...}  
}
```



The diagram consists of five horizontal lines on the right side, each with an arrow pointing to a specific annotation in the code block on the left. The labels on the right are: 'Class Invariant.' (pointing to '@@ invariant'), 'Method Precondition.' (pointing to '/*@ requires'), 'Method Postcondition.' (pointing to '@ ensures'), 'Exception raised by the method.' (pointing to '@ signals'), and 'Method Body (Java).' (pointing to the method signature 'public void boo() throws Exception {...}').

JML supports Design by Contract by means of two styles of specifications: *lightweight* specifications and *heavyweight* specifications. In this work, due to the restricted platform for which it is being developed, we are only concerned with *lightweight* specifications, where only preconditions are required for all methods, but postconditions can be omitted. Omitted postconditions are interpreted as **true** (trivially satisfied) in *lightweight* specifications.

JML annotated Java programs may be compiled with the JML compiler into instrumented Java programs. These resulting programs perform the original computation together with the verification steps. For instance, if a pre-condition p is specified for a method m and, at runtime, m is called in a context where p evaluates to false, the execution of m can be blocked and a warning message be sent to the user. Also, if in spite of all care, a specified class invariant is broken by some method execution, this situation can be immediately detected and made visible to the user. Together with static verification, this kind of verification provides confidence in applications, and so, it should be particularly useful in the smart card domain. However, because of this domain's severe restrictions, it is not yet available for it. More precisely, what is currently done is that most of this instrumentation code is manually programmed and inserted in the card programs without the required rigor.

3. Java Card

Java Card is a Java platform designed for resource-constrained devices. Due to the nature of those devices, the platform is quite limited. Memory restrictions are crucial for the design of Java Card applications: typical smart cards have as low as 12KB RAM, 374KB

ROM and 144KB EEPROM. The read-only memory is used to store the Java Card Virtual Machine (JCVM), a severely downsized version of the JVM. The main differences between JVM and JCVM standards are the exclusion of some important JVM features such as many primitive types, dynamic classes and threads.

Java Card applications are called applets. In order to run one of these applets in a Java Card device, one must: (1) write the Java Card applet; (2) compile the applet; (3) convert the binary classes into a converted applet CAP file; (4) install the CAP file on the card; (5) run the applet. Compliance problems are detected by an off-card component of the JCVM [Chen 2000], the converter, before the applet is installed on the card.

Because of these restrictions, a typical Java Card application will be very limited, and only some basic functionalities will be provided on-card. Most of the more heavy processing is executed on the so-called *host side*, the program that runs on the terminal to which the card is temporarily connected. However, for safety reasons, some card data may not be seen by the host application. Such sensible data must be manipulated on the card, safely and correctly. That is one of the advantages of smart cards with respect to magnetic strip cards: their on-card code may be used to ensure the safety and correctness of data apart from the host application. And this is where tools for a rigorous specification and verification of on-card data and functionalities becomes necessary.

4. Java Card Modeling Language (JCML)

JML has been designed for standard Java. This means that all Java constructs can be used in the generated verification code. To specify and verify Java Card applications, only those commands that can be run on the device may be used. So, the verification code generated by our specification must be Java Card compliant. Besides, time and memory consumption have to be taken into account on the definition of the modeling language and for the generation of code. The design of such a specification language has to consider the trade-off between expressiveness and feasibility.

We propose JCML: the Java Card Modeling Language [Souza Neto 2007]. JCML inherits as many JML constructs as possible. In particular, the specification part of the language is preserved. Only the Java part of JML has been pruned to cope with the restrictions of the Java Card language. The removed features include:

- Types: The primitive Java types `long`, `float`, `double` and `char`, as well as multidimensional arrays.
- Language Features: Dynamic class loading, threads, object cloning, and some aspects of package access control.
- Exceptions: Some exception and error subclasses.
- Library classes: Most of the Java core API classes and interfaces such as `Java.io`, `Java.lang` and `Java.util`. Classes such as `Boolean`, `Integer` and `String` are not supported either. The `Object` class exists, but without most of its methods.

The following list shows some of the JCML constructs:

- class and interface specifications: class invariants;
- method preconditions, normal and exceptional postconditions;
- method assignable conditions (variables that may be modified by a method);

- JML's extensions to Java's expression syntax such as quantifier keywords (`\forall` and `\exists`) and forward and reverse implication operators (`==>` and `<==`).

The complete JCML grammar is available at [Souza Neto 2007]¹, and has been produced from the JML grammar [Leavens et al. 2006] by selecting the set of rules concerned with *lightweight* specifications and removing all constructs not supported by Java Card. JCML specifications support Java Card expressions, in the same way that JML supports Java expressions. The (Java) code generated by our JCML compiler (section 5) obeys these constraints.

5. The JCML Compiler

The current version of the JCML compiler generates Java Card-compliant verification code for lightweight specifications. It implements invariant and condition verifications. The organization of the generated code preserves the structure of the original Java Card program, adding the verification code to it. The focus of the implementation is the generation of code suitable to be run on very restrictive devices. Because of this, our compiler had to be developed from scratch.

For each condition in the JCML annotations, a checking method is generated. The compiler uses the *wrapper* approach proposed in [Cheon 2003]. In this approach, the code of each (annotated) method of the program is embedded in a new method, whose task is to verify the assertions and call the original method. The embedded methods are renamed and made private. The wrapper method has the same name and signature as the original method it wraps. The wrapper method checks the preconditions and invariants and then executes the original method. After that, the wrapper checks the invariant and any specified postconditions.

In JCML one can specify static and instance invariants. Static invariants are properties over static attributes. Instance invariants deal with instance fields. Instance methods can contain both static and instance invariants. The JCML compiler generates a separate invariant method for each kind of invariant found in the source code.

According to the proposal in [Cheon 2003], some auxiliary methods are generated for the wrapper. Such methods are defined for the verification of preconditions, postconditions and invariants. The auxiliary methods rise exceptions when the assertions are violated.

In Figure 1 we present the structure of the auxiliary methods that check invariants and preconditions. Each assertion is defined by a predicate P_i . The condition verified by the auxiliary method `checkInv$ClassName$` (lines 1–7) corresponds to the conjunction of all the predicates for the invariants for the class `ClassName`. If any of the predicates P_i evaluates to *false*, an `InvariantException` is signaled. This method does not take parameters, since invariants are defined over global variables.

The `checkPre$MethodName$(T1 a1, ..., Tn an)` method (Figure 1, lines 9–15) performs a similar verification. This method has the same parameters as the verified method. The case of postconditions is analogous.

¹Available at <http://www.cefetrn.br/placido/PlacidoAntonioDeSouzaNeto.pdf>

```

1 private void checkInv$ClassName$ () {
2     if (!(P1)) {
3         ISOException.throwIt (InvariantException.SW_INVARIANT_ERROR);
4     ...
5     if (!(Pn)) {
6         ISOException.throwIt (InvariantException.SW_INVARIANT_ERROR);
7     }
8
9 private void checkPre$MethodName$(T1 a1, ..., Tk ak) {
10     if (!(P1)) {
11         ISOException.throwIt (RequiresException.SW_REQUIRES_ERROR);
12     ...
13     if (!(Pm)) {
14         ISOException.throwIt (RequiresException.SW_REQUIRES_ERROR);
15 }

```

Figure 1. Java Card methods to check invariants and preconditions.

5.1. An Example

We present a simple Java Card application named *UserAccess* (Figure 2), which runs as a card-controlled printing quota for students and staff. The application also grants access to certain parts of the building to the user. We propose a JCML specification for this application. It will be used to demonstrate how runtime verification code is generated.

The *UserAccess* class includes the following methods:

`setID (byte [] m)`: Defines the user ID.

`getID ()`: Returns the user ID.

`addArea (byte local_cod)`: Includes a new local to the array of places accessible by the user.

`hasAccess (byte local_cod)`: Return true if the user has access granted to the place identified by the parameter.

`addCredits (short value)`: Adds some printing credits to the user.

`removeCredits (short value)`: Removes a number of printing credits from the user.

`short getCredits ()`: Returns the balance of printing credits.

`setType (byte [] m)`: Defines the user type (student or professor).

`getType ()`: Returns the user type.

The file (`UserAccessJCML.java`) is generated from the JCML source containing the *UserAccess* class. The generated file contains the assertion-checking methods. The `UserAccessJCML.java` program can be compiled with a standard Java compiler in order to generate the executable class, and finally converted into the CAP file which runs on the card.

Specification of Invariants: In JCML, invariants for a class are properties that must hold throughout every instance of the class. JCML invariants are checked before and after each method is called. The only variables allowed to appear in JCML invariants are the class attributes.

The *UserAccess* invariant (Figure 2, lines 23–39) defines: (i) that no `userId` may have more than `MAX_USER_ID_LENGTH` (line 24); (ii) that every user is either a student

```

1 import Javacard.framework.*;
2 public class UserAccess {
3
4     public static final byte    MAX_USER_ID_LENGTH = 15;
5
6     //types of users
7     public static final byte    STUDENT = 0;
8     public static final byte    PROFESSOR = 1;
9
10    //different requirements for different types of users
11    public static final byte    MAX_AREAS = 20;
12    public static final short   MAX_CREDITS = 3000;
13    public static final byte    STUDENT_MAX_AREAS = 10;
14    public static final short   STUDENT_MAX_CREDITS = 1000;
15
16    //class attributes
17    private /*@ spec_public @*/ byte[]   userId;
18    private /*@ spec_public @*/ byte     userType;
19    private /*@ spec_public @*/ byte[]   authorizedAreas;
20    private /*@ spec_public @*/ byte     nextArea;
21    private /*@ spec_public @*/ short    printerCredits;
22
23    // no userId may have more than MAX_USER_ID_LENGTH
24    /*@ invariant userId.length <= MAX_USER_ID_LENGTH; @*/
25
26    //every user is either a student or a professor
27    /*@ invariant userType == STUDENT || userType == PROFESSOR; @*/
28
29    // global limits and values
30    /*@ invariant authorizedAreas.length <= MAX_AREAS; @*/
31    /*@ invariant \forall byte a; 0 <= a &&
32                a < authorizedAreas.length;authorizedAreas[a] >= 0; @*/
33    /*@ invariant printerCredits >= 0 && printerCredits <= MAX_CREDITS; @*/
34
35    //restricted limits for students
36    /*@ invariant userType == STUDENT ==> authorizedAreas.length <=
37                STUDENT_MAX_AREAS; @*/
38    /*@ invariant userType == STUDENT ==> printerCredits <=
39                STUDENT_MAX_CREDITS; @*/
40    ...

```

Figure 2. UserAccess Class (with invariants).

```

1 private void checkInv$UserAccessJCML$() throws InvariantException {
2     if (!(userId.length <=MAX_USER_ID_LENGTH )) // (i)
3         ISOException.throwIt (InvariantException.SW_INVARIANT_ERROR);
4     if (!(userType == STUDENT || userType == PROFESSOR )) // (ii)
5         ISOException.throwIt (InvariantException.SW_INVARIANT_ERROR);
6     if (!(authorizedAreas.length <=MAX_AREAS )) // (iii)
7         ISOException.throwIt (InvariantException.SW_INVARIANT_ERROR);
8     for (byte a = 0; a < authorizedAreas.length ;a++){
9         if (!(authorizedAreas [a ]>=0 )) // (v)
10            ISOException.throwIt (InvariantException.SW_INVARIANT_ERROR);
11     }
12     if (!(printerCredits >=0 && printerCredits <=MAX_CREDITS )) // (iii)
13         ISOException.throwIt (InvariantException.SW_INVARIANT_ERROR);
14     if (!(!(userType == STUDENT ) ||
15         ( authorizedAreas.length <=STUDENT_MAX_AREAS ))) // (iv)
16         ISOException.throwIt (InvariantException.SW_INVARIANT_ERROR);
17     if (!(!(userType == STUDENT ) ||
18         ( printerCredits <=STUDENT_MAX_CREDITS ))) // (iv)
19         ISOException.throwIt (InvariantException.SW_INVARIANT_ERROR);
20 }

```

Figure 3. UserAccessJCML Check Invariant Method.

or a professor (line 27); (iii) global limits and values for the number of authorized areas and printer credits per user (lines 30 and 33); (iv) restricted limits for students (lines 36 to 39); (v) that area codes are natural numbers (lines 31 and 32).

The JCML compiler translates the invariant expressions into a private invariant checking method that raises an `InvariantException` when one of the conditions is broken. The generated code is shown on Figure 3, where manual comments have been included to associate each verification to the items above. The verification code for the forall quantifier, item (iv) includes a for loop that will be explained in section 5.2.

UserAccess Methods Specification: Let us now see how a JCML-annotated method is dealt with. The `addCredits` method is used to add credits to the user. This method, shown in figure 4, has one parameter, called `value`, corresponding to the amount to be credited. The specification requires that the resulting credit balance is not greater than the allowed limit for the user, where `getCredits()` provides the current balance on the card.

The `addCredits` wrapper method (Figure 5), generated by JCML, wraps the original method call in a try-catch block that (i) checks the invariant and precondition; (ii) calls the original method and (iii) checks the invariant and postcondition. Figure 6 shows the generated code for `checkPre$addCredits$`.

5.2. Supporting Non-Java Operators

JCML includes, in its assertions, some operators that are not primitive in Java (nor in Java Card). These operators are logical implication (\implies) and universal and existential quantifiers. The JCML compiler generates the implementation of these operators in Java Card, as follows:

```

1 /*@ requires value >= 0 && (value + getCredits()) <= MAX_CREDITS &&
2     (userType == STUDENT ==>
3         (value + getCredits()) <= STUDENT_MAX_CREDITS);
4     ensures printerCredits >= value;
5 @*/
6 public void addCredits(short value) {
7     printerCredits += value;
8 }
9
10 public short getCredits(){
11     return printerCredits;
12 }

```

Figure 4. addCredits and getCredits Methods.

```

1 public void addCredits(short value) {
2     try{
3         checkInv$UserAccessJCML$ ();
4         checkPre$addCredits$(value);
5
6         addCredits$original(value); // Call the original method
7
8         checkInv$UserAccessJCML$ ();
9         checkPost$addCredits$(value);
10    }catch (InvariantException invEx) {
11        ISOException.throwIt (ISO7816.SW_CONDITIONS_NOT_SATISFIED);
12    }catch (RequiresException reqEx) {
13        ISOException.throwIt (ISO7816.SW_CONDITIONS_NOT_SATISFIED);
14    }catch (EnsuresException ensEx) {
15        ISOException.throwIt (ISO7816.SW_CONDITIONS_NOT_SATISFIED);
16    }
17 }

```

Figure 5. Generated addCredits Methods (wrapper).

```

1 private void checkPre$addCredits$( short value)
2     throws RequiresException{
3     if(!(value >=0 && (value + getCredits ())<=MAX_CREDITS &&
4         (!(userType == STUDENT ) ||
5         ( (value + getCredits ())<=STUDENT_MAX_CREDITS ))))
6         ISOException.throwIt (RequiresException.SW_REQUIRES_ERROR);
7 }

```

Figure 6. Generated method for precondition verification.

Modeling Implications: Formulas of the type $A \implies B$ found in a specification generate verification conditions corresponding to their equivalent: $\neg A \vee B$.

Modeling Quantifiers: Consider the JCML quantified expression:

```
\forall short i, j; 0 <= i && i < j && j < 10; a[i] < a[j];
```

which follows the JCML grammar rule:

```
spec-quantified-expr : ( quantifier quantified-var-decls ; [ [ predicate ] ; ]  
                        spec-expression )
```

This expression uses the universal quantifier to specify that the vector a is sorted at indexes between 0 and 9. According to the JML Reference Manual [Leavens et al. 2006], in the absence of a range predicate, the quantified expression must be evaluated over every value of the type of the quantified variable. For instance, in the previous example, the verifier would check all the possible values of i and j , from *MinShort* to *MaxShort*. Evaluating this kind of expression can be very problematic in the context of Java Card: even for short-based types, the time required for on-card verification can be unacceptable.

Our implementation tackles this problem by using static analysis to reduce the state space. The algorithm processes the conditions in a quantifier, in order to restrict the range of values assumed by each variable. Initially, our algorithm assumes the lower and upper bounds defined by [Leavens et al. 2006]. The algorithm proceeds in two steps: (1) the definition of tighter bounds for each variable and (2) the code generation.

The first step is a traversal of the quantifier's predicate. In the example predicate given above, we have the following sequence of upper and lower bounds for its variables:

1. We begin with $\text{MinShort} \leq i \leq \text{MaxShort}$ and $\text{MinShort} \leq j \leq \text{MaxShort}$.
2. From the equation $0 \leq i$ we can define a new lower bound for i . Now we have $0 \leq i \leq \text{MaxShort}$ and $\text{MinShort} \leq j \leq \text{MaxShort}$.
3. From the equation $i < j$ we can define new bounds for both i and j . Now we have $0 \leq i \leq j - 1$ and $i + 1 \leq j \leq \text{MaxShort}$.
4. Finally, from the condition $j < 10$ we can define a new upper bound for j . Now we have $0 \leq i \leq j - 1$ and $i + 1 \leq j \leq 9$.

The second step generates code to verify the specification in accordance with the bounds defined in the first step. For each variable, a (nested) for-loop is generated using the upper and lower bounds. The order in which the variables are defined is relevant. For instance, in the given example, the bounds for i must be absolute. This means that the expression $j - 1$ must be replaced by its maximum possible value 8. The bounds for j can refer to i . Our algorithm produces the following code:

```
1 private void checkMethod() {  
2   for (i = 0; i <= 8; i = i + 1) {  
3     for (j = i+1; j <= 9; j = j + 1) {  
4       if (!(a[i] < a[j])) throws Exception; }  
5   }  
6 }
```

The range predicate presented so far is a conjunction of conditions. Our algorithm will suppose that predicates in the quantifier expression are given in disjunctive normal form. For each conjunctive clause, the algorithm will produce a nested for-loop. The loops generated for each conjunctive clause will be placed in a sequence. For instance, if the code generated for a conjunctive clause P is $\mathcal{C}(P)$, the code generated for the quantified expression `\forall i, j; P or Q or S; E` will have the following structure:

```
try {
  C(P)
} catch { try {
  C(Q)
} catch {
  C(S)
}}
```

Even using our algorithm, this number of operations can be too high to be run on-card. Our JCML compiler issues a warning when such a situation arises. The user can enable the code generation for quantifiers, for instance to be used during the application test phase. This code will not be generated for the production, on-card version of the applet.

The treatment of existential quantifiers uses the equivalence $\exists x.P(x) \equiv \neg\forall x.\neg(P(x))$.

6. Experiments, Optimization and Results

In this section, the *UserAccess* example is used as a case study for our JCML compiler. Several experiments have been performed in order to evaluate the resulting code.

In order to ensure the optimal use of the resources, the JCML compiler complies with the following requirements:

- *No checking methods or calls are generated for empty specifications.* For instance, if a class does not contain an invariant specification, the method to check invariant is not generated.
- *No checking methods or calls are generated for the trivial specification (true).*
- *If there is no verification to be done for a method call, then, this method is not renamed and a wrapper for it is not defined.*
- *If the specification to be checked is a Java Card expression, then this expression is used as it is, without generating extra evaluation code for it.* This does not happens, for instance, with quantifiers.
- *Compilation flags are used to set the level of verification required.* For instance, no methods are generated for invariant or postcondition when only preconditions are required.
- *If a specification does not hold, an exception is signaled.*

The source and object code generated by our implementation is compared to the original JML implementation [Cheon 2003]. Our experiments consist on compiling the *UserAccess* example using our JCML and the original JML compilers. Our compiler

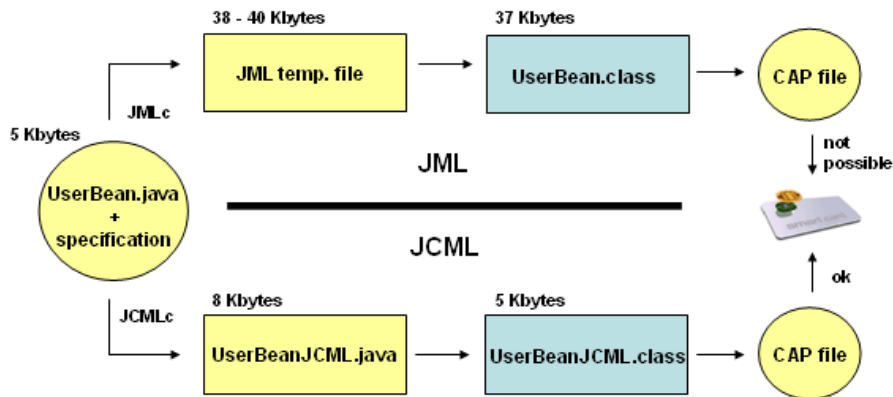


Figure 7. JMCLc x JMLc.

generates Java Card code. The code generated by the JML compiler is not Java Card compliant. We analyze both compilers in terms of the sizes of the generated code, the size of the executable class file and the execution time for each method.

Figure 7 shows the compilation process of both compilers. Both workflows are similar: the source code is translated into a Java/Java Card program, which is then compiled to obtain a CAP file. Notice that the CAP file generated from the JML specification cannot be run on-card, even if the original annotated code is Java Card compliant. This happens because the verification code generated by JMLc uses libraries and types which are not in the Java Card platform.

The wrapper approach proposed in section 5 generates methods for preconditions, postconditions, invariants and renames the original method, which becomes internal, private. This situation generates a large number of method calls.

In the following, we explore the inlining optimization technique for the JCML compiler. Method inlining can have the advantage of reducing the processing time, at the cost of possibly increasing the size of the generated code. Due to the restrictive nature of our applications, the use of this technique on the compiler should be carefully studied. We devised several versions of the compiler, using inlining to different extents:

- JCMLc 1:** No inlining. This is the wrapper approach, as presented in section 5.
- JCMLc 2:** Inlining is used only for the original methods. Instead of creating new, private methods, the body of each original method is inlined into its wrapper.
- JCMLc 3:** Inlining of original methods, pre and postconditions. Pre- and postconditions are also inlined in the wrapper methods. Invariant-checking methods are still generated.
- JCMLc 4:** All the methods are inlined into the wrapper.

Table 1 presents the number of lines of code generated for for each method of the example. The last three lines of Table 1 show the total size of the generated code, expressed in lines of Java code (**Source (LOC)**), KBytes of Java code (**.Java (KB)**) and KBytes of the executable file (**.class (KB)**). The column **Original** shows the number of lines of the original annotated program. The columns **JCMLc 1** to **JCMLc 4** show the number of lines for each version, as they are generated by our JCML compiler, using

Method	Original	JCMLc 1	JCMLc 2	JCMLc 3	JCMLc 4	JMLc
<i>setID</i>	7	46	43	31	39	294
<i>getID</i>	3	32	28	28	35	221
<i>setType</i>	8	46	43	31	39	299
<i>getType</i>	3	32	28	38	35	196
<i>addArea</i>	10	41	38	32	40	277
<i>hasAccess</i>	11	46	42	35	42	270
<i>addCredits</i>	7	38	35	31	39	262
<i>removeCredits</i>	6	38	35	29	37	268
<i>getCredits</i>	3	32	28	28	35	196
Source (LOC)	103	270	236	177	415	2394
.Java (KB)	4.78	9.05	7.83	6.08	20.04	82.2
.class (KB)	2.0	6.11	5.10	3.49	7.02	30.1

Table 1. UserAccess - Lines of code.

the optimization levels described above. The column **JMLc** contains the number of lines generated by JMLc for our example.

Notice that the number of lines of Java code generated by our implementation is much smaller than those for the equivalent JML code. For instance, for those methods without specification, such as `getCredits`, there is no additional code to be generated. In this case, our compiler copies the original code for the method, while, the JML compiler generates a large amount of code.

The code generated by JCMLc (*i*) is, in all cases, much smaller than the one generated by JMLc and (*ii*) depending on the complexity of The column **Original** shows the number of lines of the original annotated program. The columns **JCMLc 1** to **JCMLc 4** show the number of lines for each method, as they are generated by our JCML compiler, using the optimization levels described above. The column **JMLc** contains the number of lines generated by JMLc for our example. the specification, may have a size which is similar to the original annotated program, specially when we consider the executable code, which is the one effectively loaded to the card (last line of table 1).

Execution times for each method of our example are shown in Table 2. The numbers in this table are CPU times, in milliseconds. The experiment was run on a Celeron 1.3 GHz laptop with 512MB RAM. These data were collected using the *Profiler* plugin for Eclipse.

Notice that the inlining for all the generated methods resulted in execution times that are comparable to the ones without verification code, even with one quantifier in the invariant that is repeatedly checked. These execution times, together with the sizes shown in Table 1, show that the use of JCML is both possible for Java Card and not too expensive. The code generated by our compiler is consistently faster and smaller than the code generated by JMLc. The facts stated above allow us to conclude that one can afford the use of a behavioral specification language on Java Card.

Method	Original	JCMLc 1	JCMLc 2	JCMLc 3	JCMLc4	JMLc
<i>setID</i>	0.018	0.436	0.213	0.090	0.021	2.627
<i>getID</i>	0.018	0.172	0.127	0.123	0.024	2.264
<i>setType</i>	0.017	0.283	0.110	0.077	0.021	2.789
<i>getType</i>	0.018	0.129	0.077	0.076	0.019	2.223
<i>addArea</i>	0.089	0.325	0.478	0.249	0.102	5.416
<i>hasAccess</i>	0.047	0.173	0.324	0.151	0.058	2.625
<i>addCredits</i>	0.018	0.458	0.370	0.264	0.097	2.746
<i>removeCredits</i>	0.018	0.330	0.230	0.164	0.050	2.612
<i>getCredits</i>	0.017	0.453	0.301	0.315	0.075	2.431

Table 2. UserAccess - Execution times.

7. Related Work

Formal method systems that take Java Card features into account include Krakatoa [Marche et al. 2004] and the KeY System [Beckert et al. 2007b, Beckert et al. 2007a]. Krakatoa proves Java/Java Card programs annotated with JML specifications by using the *Why* [Filliâtre 2003] and *Coq* [The Coq Development Team 2007] tools. *Why* is a proof obligation generator and *Coq* is a proof assistant. Krakatoa translates Java/Java Card code into the *Why* input language (an ML-like language), which generates proof obligations to be interactively proved by means of the *Coq* proof assistant. The KeY system is intended to integrate the design, implementation and formal specification and verification of object-oriented languages. The KeY system is based on a theorem prover for the first-order Dynamic Logic for Java and can verify Java Card programs thoroughly. Both Krakatoa and KeY perform static verification only, as it is the case of other related works [Leino et al. 2000, Nimmer and Ernst 2001, Van den Berg and Jacobs 2001].

Efforts towards runtime verification of Java (and Java ME) can be found in [Rebêlo et al. 2008]. That work proposes the use of AspectJ to implement a JML compiler that takes specifications and generates bytecode compliant with both Java and Java ME virtual machines. Regarding their language constructs, Java ME is a richer language than Java Card and the architectures for which Java ME is targeted are less constrained than those in which Java Card applets run.

8. Conclusions and Future Work

This paper presented JCML – a language for specification of Java Card programs – and its associated compiler. JCML annotates Java Card programs to produce runtime verification code which can be performed on devices with severe memory and processing restrictions. JCML includes all JML constructs which can be translated into Java Card compliant code. A case study was used, and the obtained results show that the proposed approach is effective. For instance, in our example, JCML generated an executable code which is approximately 75% smaller than the one generated by JMLc, even when execution speed is the primary concern (all verification methods inlined).

The code generated by our compiler is smaller and faster than equivalent code generated by the original JCML compiler. This is due to the following facts:

- JCMLc is devised to be optimized: For instance, no tests or calls are generated for empty conditions. The original JML compiler generates code for all conditions, independently of the original JML specification.
- We use static analysis to define the upper and lower bound of variables in quantifiers.

The version of the compiler presented here is able to check some simple (yet meaningful) properties. Current work includes the extension of the class of specifications dealt with by the compiler. To be able to generate verification code for a greater class of specifications, additional optimizations need to be employed. Our next step is to deal with exceptional behaviour so that the application that runs on-card is able to gracefully recover from faults.

In another line of work, studies concerning the use of aspects in the implementation of the compiler, as done in [Rebêlo et al. 2008], will also be carried out. We plan to compare results with our current approach and to identify possible common features and improvements.

Finally, new case studies will be carried out, in order to complete the validation of the approach and to define the range of applications that can benefit from it.

References

- Abrial, J.-R. (1996). *The B Book: Assigning Programs to Meanings*. Cambridge University Press.
- Beckert, B., Giese, M., Hähnle, R., Klebanov, V., Rümmer, P., Schlager, S., and Schmitt, P. H. (2007a). The KeY System 1.0 (Deduction Component). In *CADE*, pages 379–384.
- Beckert, B., Hähnle, R., and Schmitt, P. H., editors (2007b). *Verification of Object-Oriented Software: The KeY Approach*. LNCS 4334. Springer-Verlag.
- Bhorkar, A. (2000). A Run-time Assertion Checker for Java using JML. Technical Report 00-08, Department of Computer Science, Iowa State University.
- Chen, Z. (2000). *Java Card Technology for Smart Cards: Architecture and Programmer's Guide*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.
- Cheon, Y. (2003). *A Runtime Assertion Checker for the Java Modeling Language*. PhD thesis, Department of Computer Science, Iowa State University.
- Filliâtre, J.-C. (2003). Why: a Multi-language Multi-prover Verification Condition Generation. Technical report, Université Paris-Sud, France, LRI - CNRS UMR 8623.
- Jones, C. B. and Woodcock, J. (2008). Special Edition on the Mondex Case Study. *Formal Aspects of Computing*, 20(1).
- Leavens, G. T. (2007). Tutorial on JML, the Java Modeling Language. In *ASE*, page 573.
- Leavens, G. T. and Cheon, Y. (2006). Design by Contract with JML. Draft, Available from <http://www.jmlspecs.org>.
- Leavens, G. T., Poll, E., Clifton, C., Cheon, Y., Ruby, C., Cok, D., Müller, P., Kiniry, J., and Chalin, P. (2006). *JML Reference Manual*. Draft revision 1.193.

- Leino, K. R. M., Nelson, G., and Saxe, J. B. (2000). ESC/Java User's Manual. Technical note, Compaq Systems Research Center.
- Marche, C., Mohring, C. P., and Urbain, X. (2004). The KRAKATOA Tool for Certification of JAVA/JAVACARD Programs Annotated in JML. *Journal of Logic and Algebraic Programming*, 58(1–2):89–106.
- Meyer, B. (1992). Applying "Design by Contract". *IEEE Computer*, 25(10):40–51.
- Meyer, B. (2000). *Object-Oriented Software Construction*. Prentice Hall PTR.
- Nimmer, J. W. and Ernst, M. D. (2001). Static Verification of Dynamically Detected Program Invariants: Integrating Daikon and Esc/java.
- Ortiz, E. (2005). An Introduction to Java Card Technology. Technical report, Sun Microsystems.
- Rebêlo, H., Soares, S., Cornélio, M., Lima, R., and Ferreira, L. (2008). Implementing Java Modeling Language Contracts with AspectJ. In *Proceedings of the 23rd Annual ACM Symposium on Applied Computing*, pages pp. 228–233, Fortaleza-Brazil.
- Souza Neto, P. A. (2007). JCML - Java Card Modeling Language: Definição e Implementação. Master's thesis, Programa de Pós-Graduação em Sistemas e Computação, Universidade Federal do Rio Grande do Norte.
- The Coq Development Team (2007). *The Coq Proof Assistant : Reference Manual : Version 8.1*. INRIA. available at <http://coq.inria.fr/doc-eng.html>.
- Van den Berg, J. and Jacobs, B. (2001). The LOOP Compiler for Java and JML. In Margaria, T. and Yi, W., editors, *TACAS*, volume 2031 of *Lecture Notes in Computer Science*, pages 299–312. Springer.
- Woodcock, J. and Davies, J. (1996). *Using Z: Specification, Refinement, and Proof*. Series in Computer Science. Prentice Hall International, Upper Saddle River, NJ, USA.