Automatic Certification of Java Source Code in Rewriting Logic*

Mauricio Alba-Castro[1,2], María Alpuente[1], and Santiago Escobar[1]

[1] Universidad Politécnica de Valencia, Spain.
{alpuente,sescobar}@dsic.upv.es
[2] Universidad Autónoma de Manizales, Colombia.
malba@autonoma.edu.co

Abstract. In this paper we propose an abstract certification technique for Java which is based on rewriting logic, a very general *logical and semantic framework* efficiently implemented in the functional programming language Maude. Starting from a specification of the Java semantics written in Maude, we develop an abstract, finite-state operational semantics also written in Maude which is appropriate for program verification. As a by-product of the abstract verification, a dependable safety certificate is delivered which consists of a set of (abstract) rewriting proofs that can be easily checked by the code consumer using a standard rewriting logic engine. Our certification methodology extends to other programming languages by simply replacing the concrete semantics of Java by a semantics for the programming language at hand. The abstract proof-carrying code technique has been implemented and successfully tested on several examples, which demonstrate the feasibility of our approach.

1 Introduction

As an emerging research field, code mobility is generating a growing body of scientific literature and industrial development. Proof-carrying code (PCC), originated by Necula [18, 19], is a mechanism for ensuring the safe behavior of programs that is useful for general software development, and particularly advantageous for the development of mobile code. In PCC, a program contains both the code and an encoding of an easy–to–check proof whose validity entails compliance with a predefined safety policy supplied by the code consumer. The safety certificate is automatically generated by the software producer, and then packaged along with the verified code. The crucial issues for a practical realization of PCC are: (i) the expresiveness of the language used to specify the policies, (ii) the size of the transmitted certificate, and (iii) the performance of validation at the consumer side. The main technologies commonly applied in PCC are type

* This work has been partially supported by the EU (FEDER) and the Spanish MEC, under grants TIN2004-7943-C04-01 and TIN2007-68093-C02-02, Integrated Action HA 2006-0007, LERNet AML/19.0902/97/0666/II-0472-FA, and Generalitat Valenciana GV06/285.

analysis [2, 13], and theorem proving [3, 21]. Recently, abstract interpretation has been also proposed as an enabling technology for PCC [1, 3].

Rewriting logic [16] is a flexible and expressive *logical framework* in which a wide range of logics and models of computation can be faithfully represented. It also provides an easy and inexpensive way to develop formal definitions of programming languages which are *directly executable* [17] as interpreters in a rewriting logic language such as Maude [7]. The verification of embedded and reactive systems in rewriting logic offers a good number of advantages, an important one being the maturity, generality and sophistication of the formal analysis tools available for it (see e.g. [7]).

In this paper, we develop an abstraction-based, PCC technique for the certification of Java source code which exploits the automation, expressiveness and genericity of rewriting logic. We focus on safety properties, i.e., properties of a system that are defined in terms of certain events not happening, which we characterize as unreachability problems in rewriting logic: given a concurrent system described by a term rewriting system and a safety property that specifies the system states that should never occur, the unreachability of all these states from the considered initial state allows us to infer the desired safety property. The safety policy is expressed in JML [15], a standard property specification language for Java modules. In order to provide a decision procedure, we enforce finite-state models of programs by using abstract interpretation [8]. The code consumer annotates each variable in the Java code with an abstract domain.

Our methodology is as follows. Starting from a definition of the Java semantics in rewriting logic formalized in [10], we develop an analysis technique for source code certification which is parametric w.r.t. the abstract domains. The key idea for the analysis is to test the unreachability of Java states that represent the counterpart of the safety property fulfilment using the standard Maude (breadth-first) search command, which explores the entire (finite) state space of the program. In the case when the test succeeds, the corresponding rewriting proofs demonstrating that those states are indeed never reachable are delivered as the expected outcome certificate. In order to lower the computational costs of validation and avoid specification burdens to the experts, certificates are encoded as (abstract) rewriting sequences that, together with an encoding in Maude of the abstraction, can be checked by standard reduction. As far as we know, the use of rewriting logic for the purpose of Java certification has not been investigated to date. Moreover, our methodology extends to other mainstream conventional languages or lower level languages (e.g. Java bytecode) by simply replacing the concrete semantics by a semantics for the programming language at hand (for instance, a rewriting logic semantics for Java bytecode can be found in [11]).

Our approach differs from other PCC approaches based on abstract interpretation in several aspects. With regard to the *abstraction carrying code* ACC approach of [1] for constraint logic programs, we share the high flexibility due to the parametericity on different abstract domains, the lightness of the (static analysis) proof checker on the consumer side, and the fact that both techniques are defined at the source-level (which is Ciao-Prolog in the case of ACC). However, their certificate is produced by means of a static analizer, and takes the form of a particular subset of the (fixpoint) analysis result that the consumer validates by means of a simpler abstract interpreter. Our certificate is mainly an encoding of the unreachability (abstract) rewriting proofs, which is closer to the original PCC [18, 19] where the safety certificate was a proof in first-order logic. [3] also focuses on abstract interpretation without relying on any theorem prover or type analysis tool, but their certificates take the form of strategies for reconstructing a fixpoint. Abstract interpretation is used in this case to reduce the proofs that are generated and checked by the theorem prover Coq for (a subset of) Java bytecode by a technique for fixpoint compression. It is worth noting that, in our framework, the abstract Java semantics is directly available to the code consumer, which can be verified once for all and trusted henceforth.

Let us motivate our work by focusing on some simple Java programs borrowed from the related literature, that we want to certify. A brief explanation of the JML notation used in the examples is found in Section 2.

Example 1. Consider a simple Java program, borrowed from [22], with the requirement to produce an even number as a result. We express this requirement as a safety policy in the assertion language JML by using the `ensures` clause and the operator `\result`. Namely, we require that the Java outcome is not an odd number when the execution of the method is completed.

```
static int even16()
 { /*@  Safety Specification:
     @     ensures \result % 2 != 1; @*/
      int x = 4; int y = x + 8;
      return x+y;
 }
```

A dedicated, standard verification tool for Java such as JavaFAN [11] can help verify the program above since there is only one initial state and its space state is finite. This can be done either by symbolic simulation or by explicit-state model checking of the property (specified in linear temporal logic). Unfortunately, no safety certificate would be delivered that could be inexpensively tested at the consumer side.

Example 2. Consider a more elaborated Java program together with a similar "even" safety policy required on both, the input and the output of the function.

```
static int evenOdd(int j)
{ /*@  Safety Specification:
    @    ensures ((j % 2) == 0) ==> ( \result % 2 == 0); @*/
       int u = 3; int v,z = 4;
       z += 30;
       v = u*8 + j;
       return z - v;
}
```

Here an infinite number of initial states is considered, although the search space is finite for each of them. Existing Java verification tools such as JavaFAN do not support program abstraction. Thus, for the infinite-state program of Example 2 above, JavaFAN can only be used as a semi-decision procedure to look for safety violations starting from specific initial states.

Our last example is more realistic, involving loops and conditionals, as follows.

Example 3. Consider a more realistic Java program, requiring a more involved condition on the input to ensure the fulfillment of the considered safety property. The parity of the output is again required to be "even" under a more complex "modulo 4" safety policy on the input parameter.

```
static int summation(int n)
{ /*@  Safety Specification:
    @     ensures  ((n % 4) == 0 | (n % 4) == 3)
    @              ==> ( \result % 2 == 0);      @*/
   int sum ; int i = 0;
   while (i<=n) { sum += i; i++; }
   return sum;
 }
```

Other safety properties that are routinely checked in PCC include data shape/size, bounds on resource consumption, and procedure level properties such as termination. In all these cases, PCC has the advantage to replace a (potentially) costly re-verification process by an easy–to–check proof at the consumer side. In this paper we do not address these different policies, which we consider as future work. Nevertheless, some of them are still plausible in the abstract interpretation framework and clearly not difficult to define in our setting, since all the necessary Java state elements such as memory, stacks, I/O, etc. are explicitly considered; see Section 3.

In Section 2 we briefly introduce the Java Modeling Language. In Section 3 we describe the rewriting logic semantics of Java considered in this paper and in Section 4 we present its abstract version, discussing all the difficulties that we have found and their solutions. Finally, in Section 5 we present our certification methodology, in Section 6 we demonstrate the practicality of our proposal with some experimental results, and conclude with some related work and future work in Section 7.

2 The Java Modeling Language

The Java Modeling Language [15] is a behavioral interface specification language that allows Java programmers to write specifications of Java classes, interfaces and modules without the difficulty of learning a language-independent formal specification language like OCL [5]. JML has been designed as an easily accessible specification language that combines the design by contract method and the model-based approach to specification to guarantee that a program satisfies its specification at execution time. That is, it contributes to the idea of including specifications into the code and then pre-compiling them into runtime checks embedded in the Java code. Java developers can specify with JML the functional properties of their programs in a generalization of Hoare logic, tailored to Java. As an interface specification language, JML can describe the names and static information found in Java declarations of Java modules with preconditions (in `requires` clauses), normal postconditions (in `ensures` clauses), invariants and exceptional preconditions (with the `signals` clauses), that express first-order logic statements. JML notation includes quantifiers `\forall` and `\exists` and specification-only fields and methods that allow more precise and complete specifications. As a behavior specification language, JML can also describe how the module will behave when used with assertions intermixed with the Java code. JML comes with a library with Java types that can be used for describing behavior mathematically like sets, sequences and relations. In this paper, we consider the simplest JML clauses: the `ensures` clause to indicate the result of a function expected by the code consumer and the `requires` clause to indicate any precondition on an input parameter of a function.

The JML specifications of a Java program can either be written as code annotations in Java program files or in separate files. The JML specifications as code annotations are treated like Java comments that are ignored by the compiler. The text of an annotation could be either in one line, after the marker `//@` or, in many lines enclosed between the markers `/*@` and `@*/`.

```
/*@  requires <precondition>;
  @  ensures <postcondition if no exception raised>;
  @  signals(E) <postcondition when exception E raised>;
  @  assignable <modified fields and variables>          @*/
```

3 The Rewriting Logic Semantics of Java

We assume some basic knowledge of term rewriting [20] and rewriting logic [16]. In the following, we briefly describe the rewriting logic semantics of Java given in [10] and used by the JavaFAN verification tool [11, 12]. Its novelty and interest are based on the following four advantages: (i) formal specifications provide a rigorous semantic definition for a language that can be mathematically scrutinized; (ii) such formal specifications can be developed with relatively little effort, even for large languages like Java [11] and the JVM [12]; (iii) the Maude programming language [7], which implements rewriting logic, provides a formal analysis infrastructure, so that its formal analysis tools (such as state-space breadth-first search and LTL model checking) become available for free for each programming language that is specified in Maude; and (iv) in spite of their generality, those formal analyses can be performed with competitive performance (see [11]).

The specification of Java operational semantics is a rewrite theory, that is, a triple $\mathcal{R}_{\text{Java}} = (\Sigma_{\text{Java}}, E_{\text{Java}}, R_{\text{Java}})$, with Σ_{Java} an order-sorted signature, $E_{\text{Java}} = \Delta_{\text{Java}} \uplus B_{\text{Java}}$ a set of Σ_{Java}-equational axioms where B_{Java} are axioms such as associativity, commutativity and identity and Δ_{Java} are a set of terminating and confluent (modulo B_{Java}) set of Σ_{Java}-rewrite rules, and R_{Java} a set of Σ_{Java}-rewrite rules. Intuitively, the sorts and function symbols in Σ_{Java} describe the static structure of the Java program state space as an algebraic data type, the equations in Δ_{Java} describe the operational semantics of its deterministic features, and the rules in $\mathcal{R}_{\text{Java}}$ describe its concurrent features. Following the rewriting logic framework [20, 16], we denote by $u \rightarrow^{r}_{\mathrm{Java}} v$ the fact that concrete terms u, v, denoting Java program states, are rewritten (at the top position, see [10]) by using r, which is either a rule in R_{Java} or an equation in Δ_{Java} both applied modulo B_{Java}. We simply write $u \rightarrow_{\mathrm{Java}} v$ when no confusion can arise. We denote by $\rightarrow^{*}_{\mathrm{Java}}$ the extension of $\rightarrow_{\mathrm{Java}}$ to multiple rewrite steps, i.e., $u \rightarrow^{*}_{\mathrm{Java}} v$ if there exist $u_1, \ldots, u_k$ such that $u \rightarrow_{\mathrm{Java}} u_1 \rightarrow_{\mathrm{Java}} u_2 \cdots u_k \rightarrow_{\mathrm{Java}} v$. Associativity, commutativity and unity (written ACU) axioms of binary operations in B_{Java} allow us to elegantly and effectively define (and implicitly implement) the crucial infrastructure of the Java programming language, including environments, threads, memory, input/output, synchronization information, and stores as well as the lookup operations on them; all of them implemented as a multiset union operation that builds up a "soup" of elements; see [10]. The rewrite theory $\mathcal{R}_{\mathrm{Java}}$ is defined as terms of a concrete sort State, with the main state attributes (i.e., constructors of the algebraic type State) such as `in`, `out`, `mem`, or `store`. They define an algebraic structure which is parametric on a generic sort Value that defines all the possible values returned by Java functions, or stored in the memory, etc. For instance, the `int` and `bool` constructors describe Java, integer and boolean values and are defined in Maude as "`op int : Int -> Value .`" and "`op bool : Bool -> Value .`", where `Int` and `Bool` are the internal built–in Maude sorts to define integers and booleans. Intuitively, equations in Δ_{Java} and rules in R_{Java} are used to specify the changes to the program state, i.e., the changes to the memory, threads, input/output, etc.

In [10], a sufficiently large subset of full Java 1.4 language is specified in Maude, including multithreading, inheritance, polymorphism, object references, and dynamic object allocation. However, Java native methods and many of the Java built-in libraries available are not supported. The semantics of Java is defined modularly, i.e., different features of the language are defined in separate Maude modules so to ease extensions and maintenance. See [10] for further details.

The semantics of Java is defined in a *continuation-based style*. Continuations maintain the control context of each thread, which explicitly specifies the next steps to be performed by the thread. Continuations are a typical technique to transform the uncontrollable control context into controllable data context, by stacking the sequence of actions that still need to be executed. Once the expression e on the top of a continuation (e `->` k) is evaluated, its result will be passed to the remaining continuation k. Continuations significantly ease the definition of flow-control instructions, such as break, continue, return, and exceptions. For instance, the Java addition operation on Java integers is specified[3] in Figure 1 using continuations, where `k` is the constructor symbol used to denote a continuation in a thread, `->` is the constructor symbol used to concatenate continuations,

[3] The Maude syntax is almost self-explanatory. The general point is that each item: a sort, a subsort, an operation, an equation, a rule, etc., is declared with an obvious keyword: `sort`, `subsort`, `op`, `eq`, `rl`, etc., with each declaration ended by a space and a period. We use uppercase letters to denote Maude variables and lowercase letters to denote Maude constructor symbols. See [7] for details.

```
--- First evaluate arguments
eq k((E + E) -> K) = k((E, E) -> (+ -> K)) .
--- Once arguments are evaluated to integers, compute addition
eq k((int(I), int(I)) -> (+ -> K)) = k(int(I + I) -> K) .
```

Fig. 1. Continuation-based equations for Java addition operator on integers

```
--- First evaluate arguments
eq k((E <= E') -> K) = k((E, E') -> (<= -> K)) .
--- Once arguments are evaluated to integers, compute boolean
eq k((int(I), int(I')) -> (<= -> K)) = k(bool(I <= I') -> K) .
```

Fig. 2. Continuation-based equations for Java less-or-equal operator on integers

`int` is the constructor symbol used to denote a Java integer, and `+` with[4] arity 2 and inside the constructor `int` is the Maude addition symbol, whereas `+` with arity 2 but outside the constructor `int` is the Java addition symbol, and `+` with arity 0 is a continuation symbol used to remember that the Java addition action is being stacked. The Java less-or-equal boolean operation on Java integers is specified in a similar way in Figure 2.

A relevant construction in the Java semantics is the `buildEnv` continuation symbol shown in Figure 3, that gives a new location in the memory store to each new variable. It involves the following four elements of the Java state: the thread adding new variables (denoted by constructor `t`), the environment inside the thread (denoted by constructor `env`), the store shared by all threads (denoted by constructor `store`), and a counter for the last used location in the store (denoted by constructor `nextLoc`).

Another important aspect of the semantics is the use of Java variables. In Figure 4 we show how the content of a Java variable is retrieved from the store in the Java state. The assignment operator for Java variables is specified in Figure 5. Note that the relative order among assignment and retrieval operations is relevant since multiple threads can try to concurrently assign a value to a variable or read its value from the store; hence a rule, instead of an equation, is used to represent the physical assignment as well as the physical retrieval from the store. In other words, the assignment operator and the retrieval of a variable value are non-deterministic due to the presence of different threads and are specified with Maude rules instead of Maude equations.

The state space associated to a rewrite theory is determined in Maude only by the program rules, since equations are deterministic. That is, rules and equations are applied in the same way but Maude only keeps track of the rules applied and omits the information about the equations applied. Therefore, the number of rules and equations is relevant and the smaller the number of rules, the more efficient the verification analysis, since the search space is smaller. According to [10], the Java operational semantics contains about 424 equations and only 7 rules, which considerably saves memory and execution time.

The following example illustrates the mechanization of the Java semantics.

[4] The Maude syntax allows overloading of operators, with different arities.

```
--- No new variable, end buildEnv continuation
eq k(buildEnv(noParameters, noValues) -> K) = k(K) .
--- New variable with name Var and value Val assigned to Location I' + 1
eq t(k(buildEnv(((T d(Var)), Pl), (Val, Vl)) -> K) env(Env) TC)
   store(ST) nextLoc(I')
 = t(k(buildEnv(Pl, Vl) -> K) env([Var, l(I' + 1)] Env) TC)
   store([l(I' + 1), Val] ST) nextLoc(I' + 1) .
```

Fig. 3. Continuation-based equations for building the environment

```
--- First obtain location in store from variable name
eq k(Var -> K) env([Var, Loc] env) = k(#(Loc) -> K) env([X, Loc] env) .
--- Then obtain value stored in such location
rl t(k(#(Loc) -> K) TC) store([Loc, Value] Store)
=> t(k(Value -> K) TC) store([Loc, Value] Store) .
```

Fig. 4. Continuation-based equations for variable content retrieval

Example 4. Consider the Java program of Example 1 together with the following Java main function:

```
void main() { System.out.println(addition()); }
```

The Maude command `search` provides us built–in breadth-first search, i.e., it provides all the sequences of rules (recall that the application of equations is omitted within the search space) from an initial term (without variables) to a final term (possibly with variables) [7]. Note that the initial term (without variables) describes a concrete initial Java state and the final term (possibly with variables) describes a (possibly infinite) set of final Java states. In the search command below we ask for all possible values returned by the `main` Java function of Example 1 and, therefore, a variable term denotes the goal state to be reached. Note that the code of the two Java functions `addition` and `main` is embedded within the search command, as well as the initial call to `main` (see [10] for details on how to build an initial Java state).

```
search in PGM-SEMANTICS : java((preprocess(default class 'Safe1Even1
extends Object implements none {
 (default static) int 'addition(noPara)throws(noType)
  {int d('x) = i(4) ; int d('y) = 'x + i(8) ; 12 @ return 'x + 'y ;}
 (public static) void 'main(t('String)[] d('args))throws(noType)
  {5 @ ('System . 'out . 'println < 'addition < noExp > > ;)}
})
t('Safe1Even1) . 'main < new string [i(0)] > noVal))
=>! X:ValueList .

Solution 1 (state 0)
X:ValueList --> int(16)
```

The search command returns that one unique possible Java execution trace is possible, which leads to the Java value 16 as the outcome of the Java instruction "`System.out.println(addition());`". The whole rewriting sequence leading to this Java value is also delivered by Maude.

4 The Abstract Rewriting Logic Semantics of Java

In this section, we develop an abstract version of the rewriting logic semantics of Java, described by the rewrite theory $\mathcal{R}_{\mathrm{Java}^\#} = (\Sigma_{\mathrm{Java}^\#}, E_{\mathrm{Java}^\#}, R_{\mathrm{Java}^\#})$,

```
--- First obtain location in store of the variable
--- while keeping expression in the continuation
eq k((Var = E) -> K) = k(getLocation(Var) -> (=(E) -> K)) .
--- Once the location is obtained, evaluate expression
--- while keeping location in the continuation
eq k(Loc -> (=(E) -> K)) = k(E -> (=(Loc) -> K)) .
--- Once the expression is computed, assign to location
eq k(Value -> (=(L) -> K)) = k([Value -> L] -> (V -> K)) .
--- General procedure to update a location in the shared memory
rl t(k([Value -> L] -> K) TC) store([L, Value'] ST)
=> t(k(K) TC) store([L, Value] ST) .
```

Fig. 5. Continuation-based equations and rules for Java assignment operator

$E_{\text{Java}\#} = \Delta_{\text{Java}\#} \uplus B_{\text{Java}\#}$ and its corresponding $\rightarrow_{\text{Java}\#}$ rewriting relation. Recall that the rewrite theory $\mathcal{R}_{\text{Java}}$ is defined on a generic sort Value. Our approach consists in extending $\mathcal{R}_{\text{Java}}$ (taking advantage of its modularity) by creating abstract domains as subsorts of the sort Value and adding the appropriate versions of the Java constructions and operators for the abstract domains.

An *abstract interpretation* (or abstraction) [8] of the program semantics is given by an *upper closure operator* $\alpha : \wp(\mathsf{State}) \rightarrow \wp(\mathsf{State})$, that is *monotonic* (for all $SSt_1, SSt_2 \in \wp(\mathsf{State})$, $SSt_1 \subseteq SSt_2$ implies $\alpha(SSt_1) \subseteq \alpha(SSt_2)$), *idempotent* (for all $SSt \in \wp(\mathsf{State})$, $\alpha(SSt) \subseteq \alpha(\alpha(SSt))$), and *extensive* (for all $SSt \in \wp(\mathsf{State})$, $SSt \subseteq \alpha(SSt)$). The intuition of this definition is that each Java program state $St \in \mathsf{State}$ is abstracted by its closure $\alpha(\{St\})$. Closure operators have many interesting properties. For instance, when the considered domain is a complete lattice, e.g. $\langle \alpha(\mathsf{State}), \subseteq \rangle$, each closure operator is uniquely determined by the set of its fixed points. In the context of abstract interpretation, closure operators are important because abstract domains can be equivalently defined by using them or by Galois insertions, as introduced in [9]. Let $\iota : \alpha(\wp(\mathsf{State})) \rightarrow A$ be an isomorphism. Then, given an upper closure operator $\alpha : \wp(\mathsf{State}) \rightarrow \wp(\mathsf{State})$, the structure $(\wp(\mathsf{State}), \alpha \circ \iota, \iota^{-1}, A)$ is a Galois insertion, where $\alpha \circ \iota$ and ι^{-1} are the abstraction and concretization functions, respectively (see [9] for further details).

In our approach, the code consumer can assign a different abstract domain to each variable in the Java code to obtain a finite-state model of the program. This is an important point, since a potential user of the tool only has to select some source variables to be abstracted together with the selected abstraction. A graphical interface equipped with user–friendly advisory facilities can help her in this process. Furthermore, the user could simply annotate the source code with JML assertions encoding the required safety policy so that the critical variables (together with their appropriate abstract domains) might be automatically inferred, although in this case the abstraction might be less accurate.

For the process of assigning an abstract domain to a source variable, we have a twofold situation, considering the theoretical and practical levels. On the theoretical level, we define an abstract function for each Java variable name x, e.g., $\alpha_{\mathtt{x}} : \wp(\mathsf{Int}) \rightarrow \wp(\mathsf{Int})$, and homomorphically extend those abstract functions to an abstract function $\alpha : \wp(\mathsf{State}) \rightarrow \wp(\mathsf{State})$. Indeed, for each variable x, α abstracts the values stored in the Java memory for x using $\alpha_{\mathtt{x}}$, which can be the

$$\begin{tikzcd}
 & & \texttt{top} & & \\
 & \texttt{even} \arrow[ur] & & \texttt{odd} \arrow[ul] & \\
\texttt{\#0} \arrow[ur] & \texttt{\#1} \arrow[urr] & & \texttt{\#2} \arrow[ull] & \texttt{\#3} \arrow[ul] \\
 & & \texttt{bot} \arrow[ull] \arrow[ul] \arrow[ur] \arrow[urr] & &
\end{tikzcd}$$

Fig. 6. Lattice of integers for the *mod*2 and *mod*4 abstractions

```
--- Define abstract domains
sorts Mod2 Mod4 . subsort Mod2 Mod4 < Value .
ops even odd : -> Mod2 .
op #_: Int -> Mod4 .
--- Define abstraction functions
op mod2 : Value -> Mod2 .
eq mod2(int(I)) = if (I rem 2 == 0) then even else odd fi .
op mod4 : Value -> Mod4 .
eq mod4(int(I)) = #(I rem 4) .
--- Equations for abstracting concrete values
op inAbsDomain : Qid Value -> Value .
eq inAbsDomain('x,int(I)) = mod2(int(I)) .
eq inAbsDomain('y,int(I)) = mod2(int(I)) .
eq inAbsDomain(Var,Value) = Value [owise] .
```

Fig. 7. Abstract domain and association of abstract domain to variable name

identity function if no abstract domain is selected. As mentioned before, these assignments of an abstract domain to a source variable can be inferred from the JML annotations, e.g. "`\result % 2`" or "`n % 4`", in the Java source code. The following example shows some abstract domains which are relevant for this work.

Example 5. Let us consider an abstract function that classifies Java integers into even and odd classes, i.e., $\texttt{mod2} : \wp(\texttt{int(}\mathsf{Int}\texttt{)}) \rightarrow \wp(\texttt{int(}\mathsf{Int}\texttt{)})$ where $\texttt{int(}\mathsf{Int}\texttt{)}$ denotes the Maude terms of sort Value that correspond to the Java integers. This abstraction is relevant for Examples 1, 2, and 3. We can choose the following abstract symbols $A = \{\texttt{even}, \texttt{odd}, \texttt{top}, \texttt{bot}\}$ to denote the following subsets $\texttt{top} = \texttt{int(}\mathsf{Int}\texttt{)}$, $\texttt{bot} = \emptyset$, $\texttt{even} = \{\texttt{int(}n\texttt{)} \mid n \ mod\ 2 = 0\}$, and $\texttt{odd} = \{\texttt{int(}n\texttt{)} \mid n\ mod\ 2 = 1\}$. We can even refine such abstract domain by including the abstraction for Java integers modulo 4, i.e., $\texttt{mod4} : \wp(\texttt{int(}\mathsf{Int}\texttt{)}) \rightarrow \wp(\texttt{int(}\mathsf{Int}\texttt{)})$. This abstraction is relevant for Example 3. We can have the following abstract symbols $A = \{\texttt{top}, \texttt{bot}, \texttt{\#0}, \texttt{\#1}, \texttt{\#2}, \texttt{\#3}\}$ where $\texttt{\#k} = \{\texttt{int(}n\texttt{)} \mid n\ mod\ 4 = k\}$ for $k \in \{0, 1, 2, 3\}$. The lattice induced by the relation $\subseteq$ on sets of Java integers is shown in Figure 6.

On the practical level, we have to supplement the original Java semantics with a new Maude function, called `inAbsDomain`, that records the abstract domain associated to each variable name and that it will be used in two points: when the variable is initially created in the Java memory and everytime its value is updated in the memory. For instance, Figure 7 shows the code of `inAbsDomain` for variables `x`,`y` of Example 1 according to the JML annotations, together with the Maude code for the abstract functions `mod2` and `mod4`. We also have to add a call to the `inAbsDomain` function in the `buildEnv` continuation symbol of Figure 3 and the Java assignment operator of Figure 5; all these modifications

```
--- BuildEnv modified equation
eq t(k(buildEnv(((T d(Var)), Pl), (Value, Vl)) -> K) env(Env) TC)
   store(ST) nextLoc(I')
 = t(k(buildEnv(Pl, Vl) -> K) env([Var, l(I' + 1)] Env) TC)
   store([l(I' + 1), inAbsDomain(Var,Value)] ST) nextLoc(I' + 1) .
--- Assignment modified equations
op = : Exp Qid -> Continuation . --- new definition
op = : Location Qid -> Continuation . --- new definition
eq k((Var = E) -> K) = k(getLocation(Var) -> (=(E,Var) -> K)) .
eq k(Loc -> (=(E,Var) -> K)) = k(E -> (=(Loc,Var) -> K)) .
eq k(Val -> (=(Loc,Var) -> K)) = k([inAbsDomain(Var,Val) -> Loc] -> (Val -> K)) .
```

Fig. 8. Modified continuation-based equations for building environment and Java assignment

```
--- Execute abstract mod2 values
eq k((even, even) -> (+ -> K)) = k(even -> K) .
eq k((even, odd) -> (+ -> K)) = k(odd -> K) .
eq k((odd, even) -> (+ -> K)) = k(odd -> K) .
eq k((odd, odd) -> (+ -> K)) = k(even -> K) .
--- Combine with standard integer values
eq k((int(I), Val) -> (+ -> K)) = k((mod2(int(I)), Val) -> (+ -> K)) .
eq k((Val, int(I)) -> (+ -> K)) = k((Val, mod2(int(I))) -> (+ -> K)) .
```

+	even	odd
even	even	odd
odd	odd	even

Fig. 9. Abstract definition and equations for abstract Java addition operator

are shown in Figure 8. Obviously, we have to provide abstract versions of all the Java operators in the Java semantics dealing with such kind of values, e.g., we must provide an approximation of integer addition, less-or-equal boolean operator, etc. dealing with the new abstract domains for integers. For instance, given the abstract function `mod2`, the addition operation on integers is specified in Figure 9.

In abstract interpretation, it is common to compress several computation steps into one abstract computation step, to reflect the fact that several distinct behaviors are mimicked by an abstract state. Consider for instance the Java less-or-equal operator `<=` of Figure 2 and the abstract function `mod2`. For the case of comparing two even expressions with `<=`, an (inaccurate) approximation of the result is the union of `true` and `false`, which is denoted by the symbol `top`. A naïve implementation of this idea would mean including the following equation in the abstract Java semantics $\mathcal{R}_{\mathrm{Java}^\#}$ (following the definition of operator `<=` in Figure 2):

```
eq k((even, even) -> (<= -> K)) = k(top -> K) .
```

This instrumentalization of the Java semantics to deal with abstraction implicitly means too many modifications, since completely different Java states could be generated that have to be packed together into a unique abstract state. For instance, consider a Java expression "`if eb then et else ef`" such that the expression `eb` returns `top` so that we have to represent within a single Java state both, the case when we reach a Java state continued by executing instruction `et` and also the case when we reach a Java state continued with the instruction `ef`. This would amount to a deep modification of the whole Java semantics, in order to cope with sets of Java states. Therefore, we adopt a different approach. When several $\rightarrow_{\text{Java}}$ rewrite steps are mimicked by an abstract Java state and those rewrite steps apply different rules or equations, we use concurrency at the Maude level. That is, we add rules to $R_{\text{Java}\#}$ to reflect the different possible evolutions of the system. Following this approach, the Java less-or-equal operator is defined as follows, describing that the comparison operator `<=` can return `true` or `false` indifferently:

```
rl k((even, even) -> (<= -> K)) = k(bool(true) -> K) .
rl k((even, even) -> (<= -> K)) = k(bool(false) -> K) .
```

Now, we are ready to formalize the abstract rewriting relation $\rightarrow_{\text{Java}\#}$, which intuitively develops the idea of applying only one rule or equation from the concrete Java semantics to an abstract Java state while exploring the different alternatives in a non-deterministic way. By abuse, we denote the abstraction of a rule $\alpha(\{l\}) \rightarrow \alpha(\{r\})$ by $\alpha(\{l\} \rightarrow \{r\})$.

Definition 1 (Abstract rewriting). *Let* $\alpha : \wp(\mathsf{State}) \rightarrow \wp(\mathsf{State})$ *be an abstraction. We define the approximated version of rewriting* $\rightarrow_{\text{Java}\#} \subseteq \wp(\mathsf{State}) \times \wp(\mathsf{State})$ *by:*

$$\begin{array}{l} SSt_1 \rightarrow_{\text{Java}\#} SSt_2 \textit{ using } \alpha(\{l\} \rightarrow \{r\}) \in (R_{\text{Java}\#} \cup \Delta_{\text{Java}\#}) \\ \quad \textit{iff } \forall u \in \alpha(SSt_1), \exists v \in SSt_2 \textit{ s.t. } u \rightarrow_{\text{Java}} v, \textit{using } l \rightarrow r \in R_{\text{Java}} \cup \Delta_{\text{Java}}. \end{array}$$

We denote by $\rightarrow^*_{\text{Java}\#}$ the extension of $\rightarrow_{\text{Java}\#}$ to multiple rewrite steps. The following result follows straightforwardly by monotonicity, idempotency, and extensitivity of the upper closure operator α.

Theorem 1 (Correctness & Completeness). *Let* $\alpha : \wp(\mathsf{State}) \rightarrow \wp(\mathsf{State})$ *be an abstraction. Let* $SSt_1, SSt_2 \in \wp(\mathsf{State})$*. If* $SSt_1 \rightarrow^*_{\text{Java}\#} SSt_2$*, then for all* $u \in \alpha(SSt_1)$*, there is* $v \in SSt_2$ *such that* $u \rightarrow^*_{\text{Java}} v$*. Let* $St_1, St_2 \in \mathsf{State}$*. If* $St_1 \rightarrow^*_{\text{Java}} St_2$*, then there exists* $SSt_3 \subseteq \wp(\mathsf{State})$ *s.t.* $\alpha(St_1) \rightarrow^*_{\text{Java}\#} SSt_3$ *and* $St_2 \in SSt_3$*.*

The breadth-first search for the abstract finite state system (finite due to the use of finite abstract domains) gives us a useful tool for symbolic execution, while keeping simple the modifications of the Java semantics in Maude. Actually, verification simply boils down to the exploration of all the rewriting sequences.

Example 6. Consider the Java functions `addition` and `main` of Example 4 and the abstract Java semantics shown above with the `inAbsDomain` function of Figure 7. The call to function `main` is now as follows. Note that, for the search command, the only change we need in this case is the replacement of `PGM-SEMANTICS` with `PGM-SEMANTICS-ABSTR`, since the considered Java function `addition` of Example 1 has no input parameters.

```
search in PGM-SEMANTICS-ABSTR : java((preprocess(default class 'Safe1Even1
extends Object implements none
{(default static) int 'addition(noPara)throws(noType)
  {((int d('x) = i(4) ;) (int d('y) = 'x + i(8) ;)) 12 @ return 'x + 'y ;}
(public static) void 'main(t('String)[] d('args))throws(noType)
  {5 @ ('System . 'out . 'println < 'addition < noExp > > ;)}})
t('Safe1Even1) . 'main < new string [i(0)] > noVal))
=>! X:ValueList .
```

This search command now returns the following result, meaning that exactly one abstract Java execution trace is proven, which returns the abstract value `even` as a result of the Java instruction "`System.out.println(addition());`":

```
Solution 1 (state 0)
X:ValueList --> even
```

and therefore every real execution of the Java program of Figure 1 also returns an even value, according to Theorem 1.

However, the abstraction defined in Example 5 is not accurate enough for the Java program of Example 3, as shown in the following example.

Example 7. Consider the code of Example 3 with the following function `main`:

```
void main() { System.out.println(sum(0)); }
```

We provide the following assignment of abstract domains for the variables in the Java program:

```
op inAbsDomain : Qid Value -> Value .
eq inAbsDomain('n,int(I)) = mod4(int(I)) .
eq inAbsDomain('i,int(I)) = mod4(int(I)) .
eq inAbsDomain('sum,int(I)) = mod2(int(I)) .
eq inAbsDomain(Var,V) = V [owise] .
```

When we search for all the results of the function `main`

```
search in PGM-SEMANTICS-ABSTR : java((preprocess(default class 'Safe1Even1
extends Object implements none {
(default static) int 'sum(int d('n))throws(noType)
  {(((int d('sum) ;) (int d('i) = i(0) ;)) 17 @ (while 'i <= 'n
  17 @ {(15 @ ('sum += 'i ;)) 16 @ ('i ++ ;)})) 18 @ return 'sum ;}
(public static) void 'main(t('String)[] d('args))throws(noType)
  {7 @ ('System . 'out . 'println < 'sum < i(0) > > ;)}})
t('Safe1Even1) . 'main < new string [i(0)] > noVal))
=>! X:ValueList .
```

Maude delivers the following two results

```
Solution 1 (state 2)                Solution 2 (state 5)
X:ValueList --> even                X:ValueList --> odd
```

which are useless since both, an even and an odd output value are possible. The problem is that the boolean condition `(i <= n)` returns both `true` and `false` (in a non-deterministic way) under the `mod2` and `mod4` abstraction operators in too many situations.

In order to improve accuracy, we define a new, more precise abstract domain $\mathtt{leq}^{\#}_{x,y}$ that is parametric w.r.t. two Java variable names x, y (which have different abstraction domains). For the previous example, this can be used to abstract variable `i` w.r.t. `n`. On the theoretical level, there are two abstract domains $\alpha_x, \alpha_y : \wp(\mathsf{Int}) \to \wp(\mathsf{Int})$ that are used for the values stored in the Java memory for variables x, y, respectively. The extension $\mathtt{leq}^{\#}_{x,y} : \wp(\mathsf{State}) \to \wp(\mathsf{State})$ takes those abstract domains α_x, α_y and captures also whether $x \leq y$ or $x > y$. On the practical level, we use the abstract symbols `leq#` and `gt#` defined in Maude as "`leq# : Abst Qid -> AbstLeqN`" and "`gt# : Abst Qid -> AbstLeqN`" where

<=	any value
leq#(Val,V)	true
gt#(Val,V)	false

++	
leq#(#(I),V)	leq#(mod4(I + 1),y) if y = #(I') $\wedge$ I < I'
leq#(#(I),V)	gt#(mod4(I + 1),y) if y = #(I') $\wedge$ I $\geq$ I'
gt#(#(I),V)	gt#(mod4(I + 1),y)

```
--- Two equations for the Java less-or-equal operator on integers
eq k((leq#(Val1,Var),Val2) -> <= -> K) = k(bool(true) -> K) .
eq k((gt#(Val1,Var),Val2) -> <= -> K) = k(bool(false) -> K) .
--- This equation is the core of the new abstract domain
--- The value of Var in memory has to be obtained before incrementing
ceq t(k(leq#(#(I),Var) -> ++'(Loc) -> K) env([Var, Loc'] Env) TC)
    store([Loc',#(I')] Store)
  = t(k([NewVal -> Loc] -> leq#(#(I),Var) -> K) env([Var, Loc'] Env) TC)
    store([Loc',#(I')] Store)
 if NewVal := if (I + 1 <= I') then leq#(mod4(int(I + 1)),Var)
                               else gt#(mod4(int(I + 1)),Var) fi .
--- This other equation complements the previous one
eq k(gt#(#(I),Var) -> ++'(Loc) -> K)
 = k([gt#(mod4(int(I + 1)),Var) -> Loc] -> gt#(mod4(int(I + 1)),Var) -> K) .
```

Fig. 10. Continuation-based equations for Java less-or-equal operator on integers

the first argument denotes the abstract domain for variable x (i.e., α_x) and the second argument is just y (the name of the second variable), e.g. for the previous example we will have an abstract expression for variable `i` such as `leq#(#0,'n)` denoting that the current value of variable `i` modulo 4 is 0 and that variable `i` is less or equal to variable `n`, whatever value `n` has been assigned in the execution. The appropriate version of the Java operators relevant for this new abstract domain are shown in Figure 10. Note that we cannot use the abstract domain above for the second variable instead of its name, since the value of this variable can change dynamically. Consider, for instance, the following variant of Example 3 where the loop therein contains the assignment `n -= 1`, and thus variable `n` changes in each iteration.

Example 8. Let us reconsider now Example 7. The code of function `inAbsDomain` for Example 3 is as follows, denoting that variables `i` and `n` have domains `mod4`, variable `sum` has domain `mod2` and that the relation $i \leq n$ is also represented in the abstract domain:

```
op inAbsDomain : Qid Value -> Value .
eq inAbsDomain('n,int(I)) = mod4(int(I)) .
eq inAbsDomain('i,int(I)) = leq#(mod4(int(I),'n) .
eq inAbsDomain('sum,int(I)) = mod2(int(I)) .
eq inAbsDomain(Var,V) = V [owise] .
```

When we search for solutions for the Java function `main` using the following command

```
search in PGM-SEMANTICS-ABSTR : java((preprocess(default class 'Safe1Even1
extends Object implements none {
(default static) int 'sum(int d('n))throws(noType)
  {(((int d('sum) ;) (int d('i) = i(0) ;)) 17 @ (while 'i <= 'n
  17 @ {(15 @ ('sum += 'i ;)) 16 @ ('i ++ ;)})) 18 @ return 'sum ;}
(public static) void 'main(t('String)[] d('args))throws(noType)
```

```
  {7 @ ('System . 'out . 'println < 'sum < i(0) > > ;)}})
t('Safe1Even1) . 'main < new string [i(0)] > noVal))
=>! X:ValueList .
```

we get the following unique output, meaning that exactly one abstract Java execution trace is proven, which returns the abstract value `even` as a result of the Java instruction "`System.out.println(sum(0))`":

```
Solution 1 (state 2)
X:ValueList --> even
```

This certifies that every possible Java execution starting with an integer n such that $n \ mod \ 4 = 0$ does always return an even value. Indeed, we can verify that initial calls "`System.out.println(sum(0))`" and "`System.out.println(sum(3))`" always return `even` whereas "`System.out.println(sum(1))`" and "`System.out.println(sum(2))`" return `odd`.

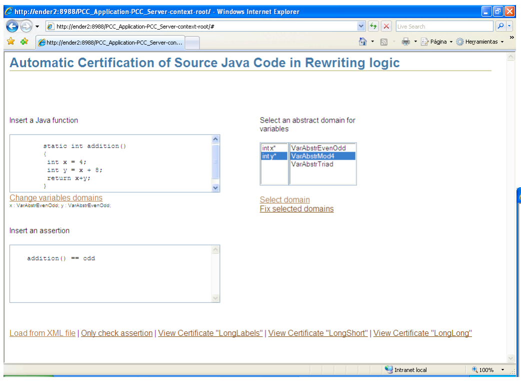
5 Certifying Java

Examples 4, 6, 7, 8 above illustrate how our methodology generates a safety certificate which essentially consists of the set of (abstract) rewriting proofs of the form $t_1 \rightarrow^{r_1}_{\mathrm{Java}^\#} t_2 \cdots \rightarrow^{r_{k-1}}_{\mathrm{Java}^\#} t_k$ that describe the program states which can and cannot be reached from a given (abstract) initial state. Since these proofs correspond to the execution of the abstract Java semantics specification, which is made available to the code consumer, the certificate can be unexpensively checked on the consumer side by any standard rewrite engine by means of a rewriting process that can be very simplified. Actually, it suffices to check that each abstract rewriting step in the certificate is valid and no other valid rewritings have been disregarded, which essentially amounts to use the matching infrastructure within the rewriting engine. Note that, according to the different treatment of rules and equations in Maude, where only transitions caused by rules create new states in the space state, an extremely reduced certificate can be delivered by just recording the rewrite steps given with the rules, while the rewritings with the equations are omitted.

The certification methodology presented here has been implemented in Maude and is publicly available at `http://www.dsic.upv.es/~sescobar/JavaACC/`. In developing and deploying the system, we fixed the following requirements: 1) define a system architecture as simple as possible, 2) make the certification service available to every Internet requestor, and 3) hide the technical details from the user. The prototype system JavaACC offers a rewriting-based program certification service, which is able to analyze safety properties of Java code which are related to the safe use of types. A snapshot of JavaACC is shown in Figures 11, 12, and 13.

6 Experiments

In Table 1, we study two key points for the practicality of our proposal: the size of the reduced versus full certificates and the relative efficiency of checking


http://ender2:8988/PCC_Application-PCC_Server-context-root/ - Windows Internet Explorer
http://ender2:8988/PCC_Application-PCC_Server-context-root/#
Live Search
Página
Herramientas
Intranet local
100%

Automatic Certification of Source Java Code in Rewriting logic

Insert a Java function

```
static int addition()
{
 int x = 4;
 int y = x + 8;
 return x+y;
}
```

Change variables domains
x : VarAbstrEvenOdd; y : VarAbstrEvenOdd;

Select an abstract domain for variables

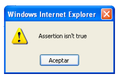
int x*	VarAbstrEvenOdd
int y*	VarAbstrMod4
	VarAbstrTriad

Select domain
Fix selected domains

Insert an assertion

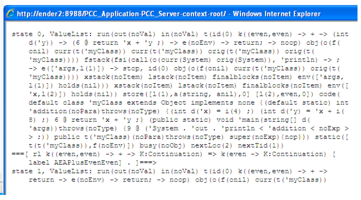
```
addition() == odd
```

Load from XML file | Only check assertion | View Certificate "LongLabels" | View Certificate "LongShort" | View Certificate "LongLong"



Assertion isn't true

Fig. 12. JavaACC Reply


http://ender2:8988/PCC_Application-PCC_Server-context-root/ - Windows Internet Explorer

```
state 0, ValueList: run(out(noVal) in(noVal) t(id(0) k((even,even) -> + -> (int
    d('y)) -> (6 @ return 'x + 'y ;) -> e(noEnv) -> return; -> noop) obj(o(f(
    onil) curr(t('myClass)) curr(t('myClass)) orig(t('myClass)) orig(t(
    'myClass)))) fstack(fsi(call(o(curr(System) orig(System)), 'println) -> ;
    -> e(['args,l(1)]) -> stop, id(0) obj(o(f(onil) curr(t('myClass)) orig(t(
    'myClass)))) xstack(noItem) lstack(noItem) finalblocks(noItem) env(['args,
    l(1)]) holds(nil))) xstack(noItem) lstack(noItem) finalblocks(noItem) env([
    'x,l(2)]) holds(nil)) store([l(1),a(string, anil),0] [l(2),even,0]) code(
    default class 'myClass extends Object implements none {(default static) int
    'addition(noPara)throws(noType) {(int d('x) = i(4) ;) (int d('y) = 'x + i(
    8) ;) 6 @ return 'x + 'y ;} (public static) void 'main(string[] d(
    'args))throws(noType) {9 @ ('System . 'out . 'println < 'addition < noExp >
    > ;)} public t('myClass)(noPara)throws(noType) super(noExp){nop}}) static([
    t(t('myClass)),f(noEnv)]) busy(noObj) nextLoc(2) nextTid(1))
===[ rl k((even,even) -> + -> K:Continuation) => k(even -> K:Continuation) [
    label AEAPlusEvenEven] . ]===>
state 1, ValueList: run(out(noVal) in(noVal) t(id(0) k((even,even) -> + ->
    return -> e(noEnv) -> return; -> noop) obj(o(f(onil) curr(t('myClass))
```

Fig. 11. JavaACC Snapshot

Fig. 13. JavaACC Certificate

Code example	Source Size (bytes)	Full Cert. Size (Kbytes)	Red Cert. Size (Kbytes)	F/R	Full Cert. Gen. Time (ms)	Red Cert. Gen. Time (ms)	Full Cert. Val. Time (ms)	Red Cert. Val. Time (ms)
even16	562	117	0.93	126	∼0	∼0	∼0	∼0
even16*	767	401	3.58	112	6	4	4	2
evenOdd	671	312	1.08	288	∼0	∼0	∼0	∼0
summation	870	1551	39.03	40	2294	146	1628	103

Table 1. Sizes of source code and certificates, and times ofcertificate generation and validation times

certificates w.r.t. their generation. The experiments have been performed on a MacBook with 2 Gb RAM. Programs even16, evenOdd, and summation are the Java programs of Examples 1, 2, and 3, respectively. Program even16* performs more involved arithmetic computations than even16, including subtraction and multiplication, while returning the same result. The first column contains the size (in bytes) of the source code for each benchmark program. The three columns for Full Cert. show the size in Kbytes, the generation time, and the validation time, respectively, for the full certificates. Similarly for the three columns of Red. Cert. Running times are given in milliseconds and were averaged over a sufficient number of iterations. Our figures demonstrate that the reduction in size of the certificate is very significant in all cases, ranging the quotient F/R (Full Cert. Size/Red. Cert. Size) from 288 in even16* to 40 for summation. When we compare the time employed to generate the (full and reduced) certificates w.r.t. the corresponding validation time, we have that the validation time is reduced by a factor up to 50%. Thus we conclude that, by minimizing the number of equations in the certificate, we achieve a simpler and indeed superior certificate that can be verified much more efficiently.

7 Conclusions and Related Work

Correctness of JML specifications can be verified either during runtime or statically. The most basic static tool support for JML is type checking and parsing (see [5]). At runtime an exception is raised if a JML condition fails.

There are several tools for static verification of Java programs using JML as specification language. The main differences between these tools regard its soundness, its level of automation, its language coverage and whether they are proof tools or just validation tools. The ESC/Java tool [14] offers a higher level of automation without any user interaction and relies on a complete (but unsound [4]) automatic prover to check null pointers or array bounds limits which uses its own specification language. The ESC/Java2 tool [6] extends ESC/Java to support more of the JML syntax and to add other functionality but it is also unsound and incomplete. It supports Java 1.4 code with JML annotations but we can not generate certificates whenever the validation succeds. Another drawback is that there is no arithmetic axiomatization that enables reasoning within ESC/Java2 about programs with integer computation [5].

As a conclusion, as far as we know our approach is the first sound and complete, fully automatic certification tool that applies to the verification of source Java code. The proposed methodology features quality attributes (notably reliability and security, but also good performance) through rigorous mechanisms which integrate a wide range of well-established programming language techniques (abstract interpretation, program semantics, meta-programming, etc). Our approach is based on a rewriting logic semantics specification of the full Java 1.4 language [10], and thus works with the full Java 1.4 language. Our certification methodology extends to other programming languages by simply replacing the concrete semantics by a semantics for the programming language at hand. Different safety policies can be defined using different (abstract) terms denoting the states that should not be reached. Such safety policies are certified by the code producer and easily checked by the code consumer using a rewriting process that can be very simplified. Certificates are encoded as (abstract) rewriting sequences which can be checked in the abstract Java semantics written in Maude on the consumer side by standard reduction. We are currently investigating how other formal verification techniques such as (abstract) model checking can be fruitfully combined with the abstraction methodology presented here to produce a more powerful methodology.

Acknowledgments. We thank Andrea Schiavinato for developing a useful web interface for our PCC tool.

References

1. E. Albert, G. Puebla, and M.V. Hermenegildo. Abstraction-carrying code. In *LPAR*, LNCS 3452, pages 380–397. Springer, 2005.
2. A. W. Appel and A. P. Felty. A semantic model of types and machine instuctions for proof-carrying code. In *POPL*, pages 243–253, 2000.
3. F. Besson, T. P. Jensen, and D. Pichardie. Proof-carrying code from certified abstract interpretation and fixpoint compression. *Theor. Comput. Sci.*, 364(3):273–291, 2006.
4. 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*, LNCS 2805, pages 422–439. Springer-Verlag, 2003.

5. L. Burdy, Y.Cheon, D. Cok, M. Ernst, J. Kiniry, G.T. Leavens, K. Rustan, M. Leino, and E. Poll. An overview of JML tools and applications. *International Journal on Software Tools for Technology Transfer*, 7(3):212–232, June 2005.
6. P. Chalin, J. R. Kiniry, G. T. Leavens, and E. Poll. Beyond assertions: Advanced specification and verification with JML and ESC/Java2. In *FMCO*, pages 342–363, 2005.
7. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. *All About Maude: A High-Performance Logical Framework.* LNCS 4350, Springer-Verlag, 2007.
8. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth ACM SYmposium on Principles of Programming Languages*, pages 238–252, 1977.
9. P. Cousot and R. Cousot. Systematic Design of Program Analysis Frameworks. In *Proc. of Sixth ACM Symp. on Principles of Programming Languages*, pages 269–282, 1979.
10. A. Farzan, F. Chen, J. Meseguer, and G. Rosu. JavaRL: The rewriting logic semantics of Java. Available at `http://fsl.cs.uiuc.edu/index.php/Rewriting_Logic_Semantics_of_Java`, 2007.
11. A. Farzan, F. Chen, J. Meseguer, and G. Rosu. Formal analysis of Java programs in JavaFAN. In Rajeev Alur and Doron Peled, editors, *CAV*, LNCS 3114, pages 501–505. Springer, 2004.
12. A. Farzan, J. Meseguer, and G. Rosu. Formal JVM code analysis in JavaFAN. In Charles Rattray, Savi Maharaj, and Carron Shankland, editors, *AMAST*, LNCS 3116, pages 132–147. Springer, 2004.
13. A. P. Felty. A tutorial example of the semantic approach to foundational proof-carrying code. In *RTA*, LNCS 3467, pages 394–406. Springer, 2005.
14. C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *PLDI*, pages 234–245, 2002.
15. G. Leavens, A. Baker, and C. Ruby. Preliminary design of JML: A behavioral interface specification language for Java. *ACM SIGSOFT Software Engineering Notes*, 31(3):1–38, 2006.
16. J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theor. Comput. Sci.*, 96(1):73–155, 1992.
17. J. Meseguer and G. Rosu. The rewriting logic semantics project. *Theor. Comput. Sci.*, 373(3):213–237, 2007.
18. G. C. Necula. Proof carrying code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Annual Symposium on Principles of Programming Languages POPL 1997, Paris, France*, pages 106–119, New York, NY, USA, 1997. ACM Press.
19. G. C. Necula and P. Lee. Safe kernel extensions without run time checking. In *Proc. of the second USENIX symposium on Operating systems design and implementation OSDI 1996*, pages 229–243, 1996. ACM Press.
20. TeReSe, editor. *Term Rewriting Systems.* Cambridge University Press, 2003.
21. M. Wildmoser, T. Nipkow, G. Klein, and S. Nanz. Prototyping proof carrying code. In Jean-Jacques Lévy, Ernst W. Mayr, and John C. Mitchell, editors, *3rd Int'll Conf. on Theoretical Computer Science (TCS2004)*, pages 333–348. Kluwer, 2004.
22. D. Wu, A. Appel, and A. Stump. Foundational proof checkers with small witnesses. In *Proceedings of the 5th ACM SIGPLAN international conference on Principles and practice of declarative programming PPDP*, pages 264–274. ACM Press, 2003.