

Replace this file with `prentcsmacro.sty` for your meeting,
or with `entcsmacro.sty` for your meeting. Both can be
found at the [ENTCS Macro Home Page](#).

A Tool for Automated Certification of Java Source Code in Maude¹

M. Alba-Castro² M. Alpuente³ S. Escobar⁴ P. Ojeda⁵
D. Romero⁶

Dpto. de Sistemas Informáticos y Computación, Universidad Politécnica de Valencia, Spain

Abstract

In previous work, an abstract certification technique for Java source code was proposed based on rewriting logic, which is a semantic framework that has been efficiently implemented in the rule-based programming language Maude. Starting from a specification of a (generic) Java abstract semantics written in Maude, we develop an abstract verification technique that essentially consists of a reachability analysis using the Java abstract semantics. We provide facilities to associate abstract domains to the variables of the considered Java program so that the resulting state-space is finite. As a by-product of the abstract verification, a safety certificate is delivered that contains a set of (abstract) rewriting proofs that can be checked by the code consumer using a standard rewriting logic engine. The main advantage is that the amount of code that must be explicitly trusted is very small. This paper presents a Web tool that implements the abstract certification technique by providing appropriate abstract domains for different safety properties while hiding the technical details of the method from the user. The tool has been devised to be easily extendable to other properties and domains. It currently supports the certification of two kinds of safety properties that are not handled by standard Java compilers: secure integer arithmetic rules and non-interference policies.

Keywords: automated certification, rewriting logic, Maude, web tool

1 Introduction

Proof-Carrying Code (PCC) is a technique that can be used for safe execution of untrusted code [13,14]. In a typical instance of PCC (see Figure 1), a code receiver (the code consumer) specifies a set of safety requirements (a safety policy) that guarantee the safe behavior of programs, and the code producer creates a formal safety proof (the

¹ This work has been partially supported by the EU (FEDER) and the Spanish MEC/MICINN, under grant TIN2007-68093-C02-02, Integrated Action Hispano-Alemana A2006-0007, the Generalitat Valenciana under grant GVPRE/2008/113, and UPV-VIDI grant 3249 PAID0607. Mauricio Alba-Castro is supported by LERNet AML/19.0902/97/0666/II-0472-FA, Pedro Ojeda is supported by the Generalitat Valenciana under FPI grant BFPI/2007/076, and Daniel Romero is supported by the MEC/MICINN under FPI grant BES-2008-004860.

² Email: malba@dsic.upv.es

³ Email: alpuente@dsic.upv.es

⁴ Email: sescobar@dsic.upv.es

⁵ Email: pojeda@dsic.upv.es

⁶ Email: dromero@dsic.upv.es

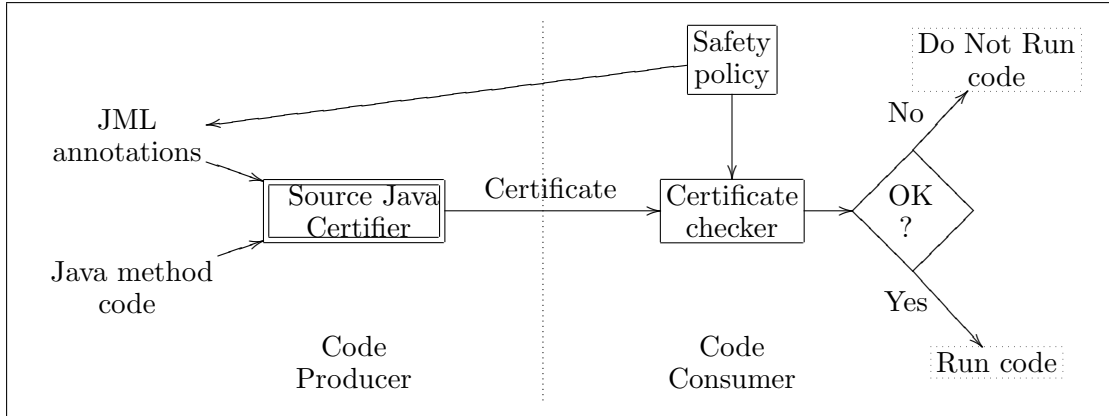


Fig. 1. Java PCC Infrastructure

certificate) that proves, for the untrusted code, compliance with the safety policy. Then, the receiver is able to use a simple and fast proof validator (the certificate checker) to check with certainty that the proof is valid and, hence, the untrusted code is safe to execute.

In [1], we presented an abstract PCC methodology for certifying Java source code that is based on rewriting logic [12]. Starting from a specification of the Java semantics written in Maude, we provide facilities to associate abstract domains to the variables in the untrusted Java program so that the resulting state-space becomes finite. This allows us to implement an abstract verification methodology that essentially consists of a reachability analysis using our abstract Java semantics. As a by-product of the abstract verification, a dependable safety certificate is delivered, which consists of a set of (abstract) rewriting proofs. This simple technique allows us to certify different safety properties that are not handled by any standard Java compiler, such as secure integer arithmetic [1] and non-interference policies [2].

This paper describes a Web tool that implements the abstract certification technique of [2,1]. The tool provides appropriate abstract domains for different properties while hiding the technical details of the method from the user. It allows safety properties to be certified, 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. Following the PCC infrastructure depicted in Figure 1, our tool can be used at the code producer side to certificate Java source code. We summarize its design principles as well as the representation issues of the safety policy and proofs, and show that the *trusted computing base* (TCB) in such a system can indeed be very small.

This paper is a new, totally redesigned implementation of the technique of [2] in order to: (i) deal with non-interference policies, (ii) improve both functionality and performance upon the former version, and (iii) make the system easier to use and extend.

2 Abstract Certification Methodology

In order to lower the computational costs of verification and avoid specification burdens on the experts, we enforce finite-state models of programs by using abstract interpretation [4]. Our methodology, which was presented in [1], can be summarized as follows. Starting from the specification of the Java semantics in rewriting logic formalized in [7], we developed an abstract, finite-state operational semantics (in rewriting logic), which is parametric w.r.t. the abstract domain and which is appropriate for program verification. Using this abstract, finite-state operational semantics, we define a source code certification technique. The key idea for the certification tool 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 that explores the abstract finite state-space of the program. In the case when the test succeeds, the corresponding rewriting proofs, which demonstrate the unreachability of those states, are delivered as the expected outcome certificate. Certificates are encoded as (abstract) rewriting sequences that, together with the extended abstract semantics and the encoding in Maude of the abstraction, can be checked by a standard reduction engine by means of a rewriting process which is very simplified.

We use the rewriting logic semantics of Java given in [7] and used by the JavaFAN verification tool [6,8]. This semantics copes with a sufficiently large subset of the full Java 1.4 language 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 novelty and interest of this semantics are based on the following 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 effort⁷, even for large languages like Java [6] and the JVM [8]; (iii) the Maude programming language [3], 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 [6].

The safety policy is expressed by means of a set of abstract requirements on the variables of the Java source program. These are written by using the Java Modeling Language JML syntax. JML is a behavioral interface specification language that accepts Java built-in operators [11]. The code consumer states the safety policy by using the JML **assert**, **requires** and **ensures** clauses intermixed with the Java code method to be certified. The text of a JML annotation can be either in one line, after the marker `//@`, or in many lines enclosed between the markers `/*@` and `@*/`.

```
/*@ requires <method precondition>;
   @ ensures <method postcondition>;
   @ assert <predicate>;           @*/
```

The tool automatically generates the Maude encoding of the abstraction, as well as the search command containing the initial state that includes the wrapped, supple-

⁷ See the different programming languages available at [7].

mented Java program, together with the final state, which depends on the expected method outcome.

The tool can generate three different certificates: *full certificate*, *reduced rules certificate*, and *reduced labels certificate*. The *full certificate* includes all Maude equations and rules (together with the corresponding matching substitutions) used in the rewriting proof. According to the different treatment of rules and equations in Maude, an extremely *reduced rules certificate* can be delivered by just recording the rewrite steps given with the rules, while the rewritings with the equations are omitted. This is justified by the fact that, in Maude, reducing with equations is deterministic and also because Maude is very efficient at doing it. Finally, the *reduced labels certificate* only records the labels of the applied rules.

The tool can certify code that complies with user-defined integer-arithmic safety properties that are not checked by standard Java compilers [1]. The tool can also certify code that obeys user-defined policies for input-output data non-interference [15]. Informally, non-interference means that the public output data do not depend on secret input data. We consider non-interference policies that associate confidentiality level labels to variables [5,9,10] (**High** labels to secret variables and **Low** labels to public non-secret variables) so that flows from variables labeled **High** to variables labeled **Low** are forbidden.

Example 2.1 Consider the following Java method, borrowed from [10], that contains a conditional statement. Its input variables `high` and `low` are respectively labeled with **High** and **Low** confidentiality labels, meaning that the variable `high` is secret and the variable `low` is public. This Java function does comply with the non-interference policy in spite of the fact that an illicit and indirect information flow, within the guard of the `if` statement, exists from the variable `high` to `low`. This is a temporary breach, because of the legal and direct flow from the constant 0, indeed a public value, to the variable `low` after the `if` statement.

```
static int mE3(int high,int low) {
    /*@ requires AbsValue(low) == #Low && AbsValue(high) == #High;
       @ ensures AbsValue(\result) == #Low;                               @*/
    if (high == 1)
        low = 1;
    low = 0;
    return low;
}
```

The abstract domain for the non-interference certification of this example is the set of confidentiality labels $LValue = \{\#High, \#Low\}$ [2]. The JML clause “requires AbsValue(high) == #High && AbsValue(low) == #Low;”, states that the variable `high` is labeled **High**, so that it is secret, and that the variable `low` is labeled **Low**, so that it is public. The clause “ensures AbsValue(\result) == #Low;” states that the outcome of the method should be public.

Let us now describe the abstract domains that we have included in our tool for certifying Java programs that make heavy use of integer-arithmic. These include, among others, the abstract domains *EvenOdd* (the abstract domain of even and odd integer numbers) and *Mod4* (the abstract domain of Java integers modulo 4). Namely,

$EvenOdd = \{\text{bot}, \#even, \#odd, \text{top}\}$, where $\text{bot} = \emptyset$, $\#even = \{\text{int}(n) \mid n \bmod 2 = 0\}$, $\#odd = \{\text{int}(n) \mid n \bmod 2 = 1\}$, and⁸ $\text{top} = \text{int}(\text{Int})$. On the other hand, $Mod4 = \{\text{bot}, \#0, \#1, \#2, \#3, \text{top}\}$, where $\#k = \{\text{int}(n) \mid n \bmod 4 = k\}$ for $k \in \{0, 1, 2, 3\}$. The JML clause “requires AbsValue(n) == #even;” states that the variable n is associated to the abstract domain $EvenOdd$ and its value should be even. The clause “ensures AbsValue(\result) == #even;” states that the outcome value of the method (denoted by the keyword result) should be even. The clause “assert AbsDomain(k) == EvenOdd;” states that the local variable k is associated to the abstract domain $EvenOdd$.

We also provide several combined abstract domains that are parametric w.r.t. two Java variables x, y and capture Boolean relations on Java integers. As a proof of concept we consider two Boolean relations: the less-than-or-equal-to, and the greater-than relations, i.e., $x \leq y$ and $x > y$. For instance, the JML clause “assert AbsDomain(i) == (EvenOdd, (<=, n));” states that variable i is abstracted using the domain $EvenOdd$ and that its value is less than or equal to the value of n .

Example 2.2 Consider the following Java program with a *while* statement, requiring a condition on the input to ensure the fulfillment of the considered safety property. The parity of the output is required to be **even** under the assumption that the input parameter n belongs to the abstract domain $Mod4$. $EvenOdd$ is the abstract domain associated to variable sum . In order to deal with the condition of the *while*, we have to use a combined domain for variable i : the JML clause “assert AbsDomain(i) == (Mod4, (<=, n));”, states that the variable i is abstracted using the domain $Mod4$ and that its value is less than or equal to the value of n .

```
static int summation(int n) {
    /*@   requires AbsValue(n) == #0 || AbsValue(n) == #3;
       @   ensures AbsValue(\result) == #even ;           @*/
    int sum = 0 ;
    /*@   assert AbsDomain(sum) == EvenOdd ;
    int i = 0;
    /*@   assert AbsDomain(i) == (Mod4, (<=, n)) ;
    while (i<=n) {
        sum += i;
        i++;
    }
    return sum;
}
```

In [1], the reader can find many examples of Java programs and integer arithmetic-based safety properties, together with some figures concerning the size of the certificates as well as the certificate generation and validation times.

⁸ Recall that Int is the Maude sort of integer number whereas int is the Java integer constructor used by the Maude, Java Rewriting Logic semantics.

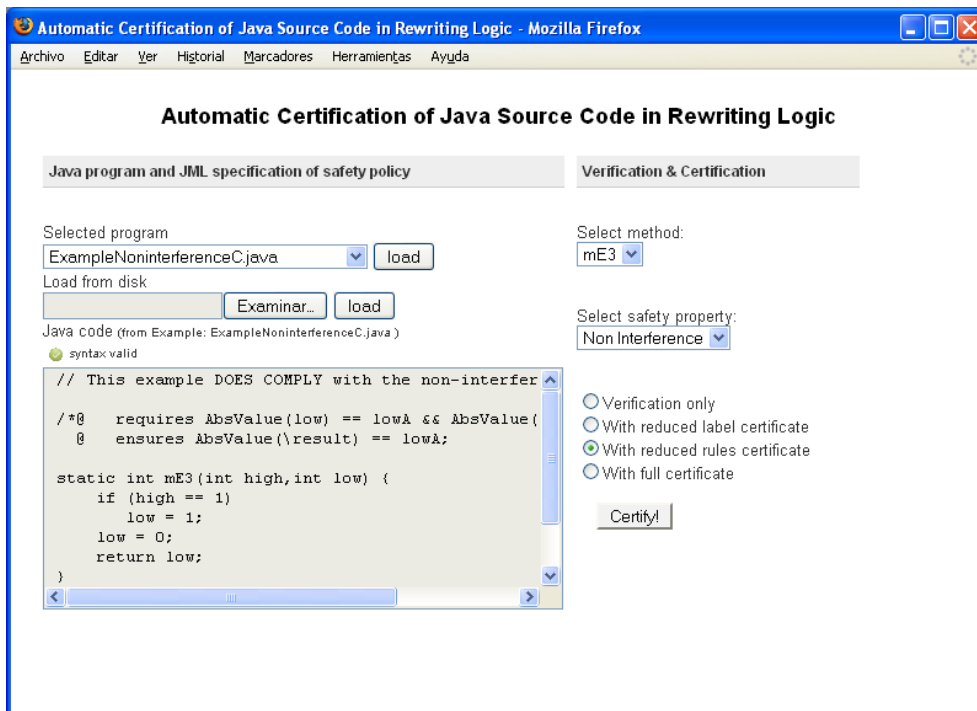


Fig. 2. Web interface of the JavaCC tool

3 Interface and Functionality

The abstract certification technique presented so far has been implemented in Maude [3]. Java has been used to develop the Web interface.

This new tool allows the user to enter the Java code methods annotated with the JML clauses, whereas the preliminary implementation of the tool [1] forced the user to associate code variables with the abstract domains by using push buttons and filling in text boxes. By using the former tool of [1], the user also had to write the method invocation with its parameter values and the method abstract outcome into text boxes. The new tool avoids such hand-writing and generates the method invocation automatically once the user selected the method to be certified.

The Web tool is publicly available together with a set of examples at <http://www.dsic.upv.es/users/elp/toolsMaude/rewritingLogic.html>. Some snapshots of the Web tool are shown in Figures 2, 3 and 4. Figure 2 shows the main page with the code of Example 2.1. Figures 3 and 4 show the corresponding reduced rules certificate split into two parts: Figure 3 shows the initial and final states, and Figure 4 shows the rewriting sequence with all the states and rules used. The main features of the Web tool are:

- The Maude programming language, which implements rewriting logic, provides a formal analysis infrastructure (such as state-space breadth-first search) with competitive performance (see [6]). Note that Maude could also be used as the infrastructure required for the proof validation process at the consumer side. 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 using the matching infrastructure within the rewriting engine. This is simple, trustworthy, and based on



Fig. 3. Reduced rules certificate: Maude search command and final state

well-understood engineering and mathematical principles.

- The Java operational semantics in rewriting logic that we have used is modular and has 2635 lines of code in 4 files [7]. We have modified 15 of the 1527 lines of code in the main file of the original Java semantics. The abstract operational Java semantics was developed as a source-to-source transformation in rewriting logic, and consists of 608 lines of extra code. This amounts to saying that, in our current system, the *trusted computing base* (TCB) is less than half the size of the original Java semantics (at least one order of magnitude smaller than the standard rewriting infrastructure and even much smaller than other PCC systems).
- In order to automate the processing of JML clauses, the tool uses the JavaCC compiler construction tool, together with a subset of the JML grammar, to generate both

```

state 0, Output: run(out(noOutput) in(noVal) t(id(0) k((High,Low) -> == -> if(7
0 ('low = i(1) ), nop) -> restoreEnv(Low) -> ((8 0 ('low = i(0) ;)) 9 0
return 'low ;) -> e(['high,1(10)] ['low,1(11)]) -> 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(['arg,1(1)]) -> stop, id(0) obj(o(f(onil) curr(t('myClass)) orig(t(
'myClass)))) xstack(noItem) lstack(noItem) finalblocks(noItem) env(['arg,1(
1) ['even,1(2)] ['high,1(9)] ['low,1(8)] ['n0,1(4)] ['nl,1(5)] ['n2,1(6)]
['n3,1(7)] ['odd,1(3)]) holds(nil) lenv(Low))) xstack(noItem) lstack(
noItem) finalblocks(noItem) env(['high,1(10)] ['low,1(11)]) holds(nil)
lenv(Low)) store([1(1),a(string, anil),0] [1(2),Low,0] [1(3),Low,0] [1(4),
Low,0] [1(5),Low,0] [1(6),Low,0] [1(7),Low,0] [1(8),Low,0] [1(9),High,0] [
1(10),High,0] [1(11),Low,0]) code(default class t('myClass) extends Object
implements none ((default static) int 'mE3((int d('high)),(int d(
'low)))throws(noType) ((8 0 (if 'high == i(1) 7 0 ('low = i(1) ;)) else nop
fi)) (8 0 ('low = i(0) ;)) 9 0 return 'low ;) (public static) void 'main(
string[] d('arg))throws(noType) ((int d('even) = i(0) ;) (int d('odd) = i(
1) ;) (int d('n0) = i(0) ;) (int d('nl) = i(1) ;) (int d('n2) = i(2) ;) (
int d('n3) = i(3) ;) (int d('low) = i(0) ;) (int d('high) = i(1) ;) 25 0 (
'System . 'out . 'println < 'mE3 < 'low, 'high > > ;)) public t('myClass)(
noPara)throws(noType) super(noExp)(nop)) static({t(t('myClass)),f(noEnv))
busy(noObj) nextLoc(11) nextTid(1)}
]==>
state 1, Output: run(out(noOutput) in(noVal) t(id(0) k(sys(println) -> callSys(
o(curr(System)) orig(System)), Low) -> ; -> e(['arg,1(1)]) -> stop) obj(o(f(
onil) curr(t('myClass)) orig(t('myClass)))) fstack(noItem) xstack(noItem)
lstack(noItem) finalblocks(noItem) env(['arg,1(1)] ['even,1(2)] ['high,1(
9)] ['low,1(8)] ['n0,1(4)] ['nl,1(5)] ['n2,1(6)] ['n3,1(7)] ['odd,1(3)])
holds(nil) lenv(Low)) store([1(1),a(string, anil),0] [1(2),Low,0] [1(3),
Low,0] [1(4),Low,0] [1(5),Low,0] [1(6),Low,0] [1(7),Low,0] [1(8),Low,0] [1(
9),High,0] [1(10),High,-3] [1(11),Low,-3]) code(default class t('myClass)
extends Object implements none ((default static) int 'mE3((int d('high)),(
int d('low))throws(noType) ((8 0 (if 'high == i(1) 7 0 ('low = i(1) ;))
else nop fi)) (8 0 ('low = i(0) ;)) 9 0 return 'low ;) (public static) void
'main(string[] d('arg))throws(noType) ((int d('even) = i(0) ;) (int d('od
d) = i(1) ;) (int d('n0) = i(0) ;) (int d('nl) = i(1) ;) (int d('n2) = i(2)
;) (int d('n3) = i(3) ;) (int d('low) = i(0) ;) (int d('high) = i(1) ;) 25 0 (
'System . 'out . 'println < 'mE3 < 'low, 'high > > ;)) public t('myClass)(
noPara)throws(noType) super(noExp)(nop)) static({t(t('myClass)),f(noEnv))
busy(noObj) nextLoc(11) nextTid(1)}
]==>
state 2, Output: pl(Low)

```

Fig. 4. Reduced rules certificate: rewriting sequence

the encoded Maude abstraction and the Maude search command, which contains the supplemented Java program with the main call that invokes the method to be certified. In order to build the initial state, it also uses the Java wrapper program that is available at [7] to transform the supplemented Java program into a Maude term.

- The Web interface allows us to both make the abstract certificate technique publicly available on-line and hide the technical details to any possible user.

The delivered certificate is generated in two steps: (i) loading the Java source code with the JML annotations, and (ii) performing the abstract reachability analysis. Let us illustrate the Web tool functionality by means of Example 2.1.

i. Program loading and fixing abstract domains. The Java code can be loaded from a file containing the Java program and the JML annotations. Additionally, the Web tool lets the user select one of several predefined examples. The file “Safe1NonInterferenceC.java”, which corresponds to Example 2.1, is a predefined example shown in Figure 2. Loading a Java program indirectly provides validation of its syntax using a standard Java compiler. The JML annotations are also validated within the generated JavaCC module.

ii. Abstract reachability analysis. The safety certificate (either full or reduced) is automatically generated by the tool by clicking on the “Certify!” button. The generation process is internally performed in the following way, shown in Figures 5, 6 and 7: (i) a new Java class is created from the Java code, with the selected method invocations and the needed parameter values, following the JML `requires` clause; the Java wrapper transforms the new Java class into a Maude term which contains the initial state (see [7] for details on how to build an initial Java state); and the

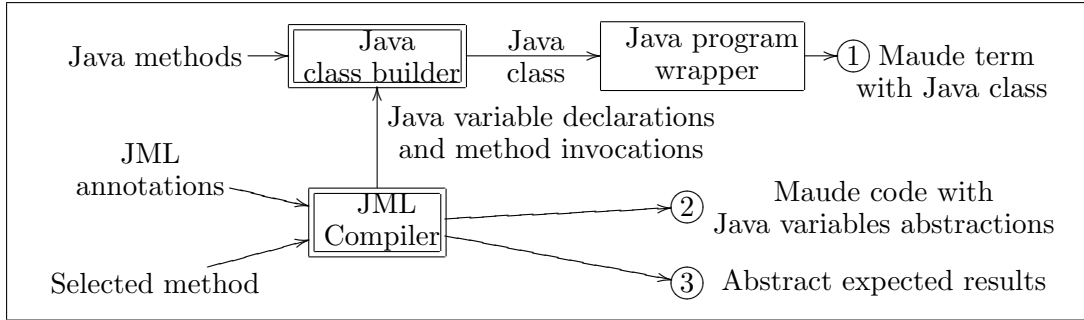


Fig. 5. JavaCC JML compiler and Java program builder

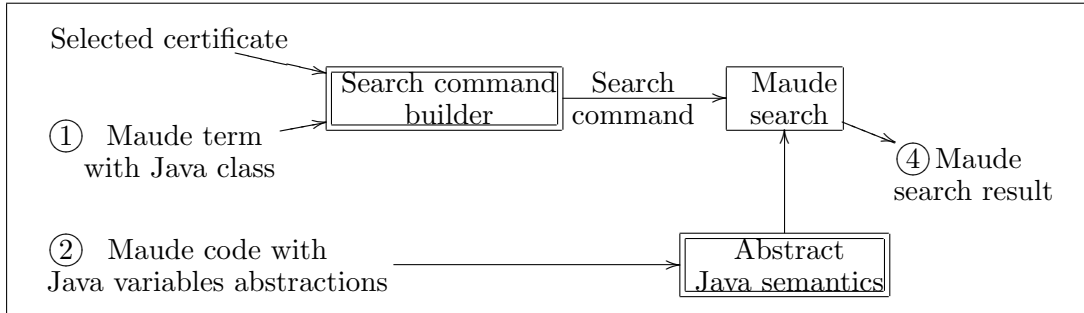


Fig. 6. Abstract Maude search

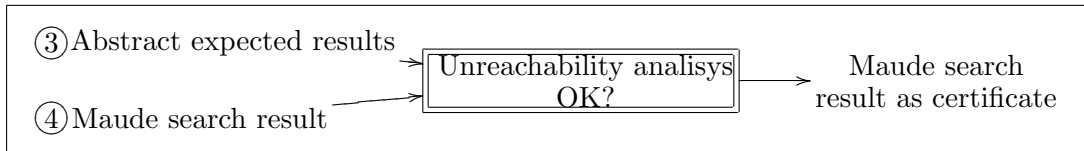


Fig. 7. Abstract Java Certification

Maude abstractions of the program variables are generated from the JML `requires` and `assert` clauses (Figure 5); (ii) we build the `search` command with the file generated by the Java wrapper, then we add the Maude abstractions of the variables to the extended abstract Java semantics, and we run the `search` file in Maude (Figure 6); and (iii) check if the final state reached is the expected result specified with the `ensures` clause, and if so, get the rewriting sequence as the certificate that will be delivered to the user (Figure 7).

An example of a reduced rules certificate is also shown in Figures 3 and 4.

4 Conclusions

Proof-carrying code has a number of technical advantages over other approaches for the problem of mobile code security. One of the most important advantages is that the trusted code of such a system can be made small. In this work, we have presented a Web tool for automated certification of Java source code developed in Maude and have shown that the trusted code can be orders of magnitude smaller than in other competing systems (e.g., Java Virtual Machines). We have also analyzed the representation issues of the safety specification and shown how they relate to the size of the abstract semantics and proof checker. In our system, the trusted code itself is based on a well-understood

and well-analyzed logical framework, which adds on to our confidence in its correctness. As far as we know, our tool is the first sound and complete, fully automated certification tool that applies to the verification of source Java code using rewriting logic.

Our tool is based on a rewriting logic semantics specification of the full Java 1.4 language that is available at [7], and thus works with the full Java 1.4 language.

References

- [1] Mauricio Alba-Castro, María Alpuente, and Santiago Escobar. Automatic certification of Java source code in rewriting logic. In *12th International Workshop on Formal Methods for Industrial Critical Systems (FMICS 2007)*, volume 4916 of *Lecture Notes in Computer Science*, pages 200–217. Springer, 2008.
- [2] Mauricio Alba-Castro, María Alpuente, and Santiago Escobar. Automated certification of non-interference in rewriting logic. In *13th International Workshop on Formal Methods for Industrial Critical Systems (FMICS 2008)*, *Lecture Notes in Computer Science*. Springer, 2009. To appear.
- [3] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn Talcott. *All About Maude: A High-Performance Logical Framework*, volume 4350 of *Lecture Notes in Computer Science*. Springer-Verlag, 2007.
- [4] Patrick Cousot and Radhia 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 (POPL 1977)*, pages 238–252, 1977.
- [5] Dorothy E. Denning and Peter J. Denning. Certification of programs for secure information flow. *Commun. ACM*, 20(7):504–513, 1977.
- [6] Azadeh Farzan, Feng Chen, José Meseguer, and Grigore Rosu. Formal analysis of Java programs in JavaFAN. In Rajeev Alur and Doron Peled, editors, *Proceedings of Computer-aided Verification (CAV'04)*, volume 3114 of *Lecture Notes in Computer Science*, pages 501–505. Springer, 2004.
- [7] Azadeh Farzan, Feng Chen, José Meseguer, and Grigore Rosu. JavaRL: The rewriting logic semantics of Java. Available at <http://fsl.cs.uiuc.edu/index.php/RewritingLogicSemanticsOfJava>, 2007.
- [8] Azadeh Farzan, José Meseguer, and Grigore Rosu. Formal JVM code analysis in JavaFAN. In Charles Rattray, Savi Maharaj, and Carron Shankland, editors, *AMAST*, volume 3116 of *Lecture Notes in Computer Science*, pages 132–147. Springer, 2004.
- [9] Sebastian Hunt and David Sands. On flow-sensitive security types. In *Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages, POPL'06*, pages 79–90, 2006.
- [10] Bart Jacobs, Wolter Pieters, and Martijn Warnier. Statically checking confidentiality via dynamic labels. In *Proceedings of the 2005 workshop on Issues in the theory of security (WITS '05)*, pages 50–56, 2005.
- [11] Gary Leavens, Albert Baker, and Clyde Ruby. Preliminary design of JML: A behavioral interface specification language for Java. *ACM SIGSOFT Software Engineering Notes*, 31(3):1–38, 2006.
- [12] José Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.
- [13] George 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.
- [14] George C. Necula and Peter Lee. Safe kernel extensions without run time checking. In *Proceedings of the second USENIX symposium on Operating systems design and implementation (OSDI 1996), Seattle, Washington, United States*, pages 229–243, New York, NY, USA, October 1996. ACM Press.
- [15] Andrei Sabelfeld and Andrew C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, 2003.