Automated Certification of Non-Interference 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 a certification technique for non-interference of Java programs based on rewriting logic, a very general *logical* and *semantic framework* efficiently implemented in the high-level programming language Maude. Non–interference is a semantic program property that prevents illicit information flow to happen. Starting from a basic specification of the semantics of Java written in Maude, we develop an information–flow extension of this operational Java semantics which allows us to observe non-interference of Java programs. Then we develop in Maude an abstract, finite-state version of the information-flow operational semantics which supports finite program verification. As a by–product of the verification, a certificate of non-interference 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.

1 Introduction

In the last decade, we have observed an increasing interest in formal methods designed for trusting code coming from untrusted sources. Proof-carrying code (PCC), originated by Necula [26], is a mechanism for ensuring the secure 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 security policy supplied by the code consumer. The security certificate is automatically generated by the software producer.In [1] we proposed an abstract PCC methodology for certifying Java source code that is based on rewriting logic. *Rewriting logic* [22] is a flexible and expressive *logical framework* in which a wide range of logics and models of computation can be faithfully represented. The methodology of [1] is as follows. Consider a

* This work has been partially supported by the EU (FEDER) and the Spanish MEC/MICINN under grant TIN 2007-68093-C02-02, Integrated Action HA 2006-0007, LERNet AML/19.0902/97/0666/II-0472-FA, and Generalitat Valenciana GVPRE/2008/113.

D. Cofer and A. Fantechi (Eds.): FMICS 2008, LNCS 5596, pp. 182–198, 2009.

(concurrent) Java program together with a specification of the Java semantics, given as a term rewriting system. Given a safety property (i.e. a system property defined in terms of certain events not happening), the unreachability of the system states denoting the situation that should never occur allows us to infer the desired safety property. Unreachability analysis is performed using the standard Maude (breadth-first) search command, which explores the entire (finite) state space of the program. In the case when the unreachability test succeeds, the corresponding rewriting proofs demonstrating that those states cannot be reached are delivered as the expected outcome certificate. Certificates are encoded as (abstract) rewriting sequences that, together with an encoding in Maude of the abstraction, can be checked by standard reduction. 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 [15].

In this paper, we extend the methodology of [1] to certify *confidentiality* by analysing *non-interference*. Confidentiality is a property by which information related to an entity or party is not made available or disclosed to unauthorized individuals, entities, or processes. However, an authorized accessing program can, on purpose or not, leak secret data in some improper way. To ensure that the program does not disclose secret data and fulfills *data confidentiality policies*, it is necessary therefore to analyse and control how information flows within the program. In this paper we focus on data confidentiality certification of Java programs. In order to express the non–interference safety policies for ensuring confidentiality, we use standard JML [21], a property specification language for Java modules. Each variable in the Java code is annotated with a confidentiality label that represents the confidentiality level of the variable and its data values.

The contributions of this paper are as follows:

- Starting from a basic specification of the semantics of Java written in Maude [14], we develop an information–flow extension of such an operational Java semantics which allows us to observe non-interference of Java programs, and is also written in Maude. For the best of our knowledge, a clear-cut semantics for Java programs dealing with non-interference was lacking. Much of previous work on ensuring Java non–interference has focused on enforcing it by appropriate information flow type systems by certifying and type preserving compilers [24,25] or bytecode typechecking [6].
- We provide an abstract, finite-state version of the information-flow operational semantics which supports finite program verification. Thanks to the different handling of rules and equations in Maude we do not suffer the state–space explosion of more traditional approaches (see [23]).
- Our Java certification methodology allows us to deal with some Java features not considered in the related literature ([20,30]): object fields, local

variables and arrays. We deal with values delivered by a `return` statement, a case not considered in [12,2,20,19]. We also consider `return` and `break` statements within conditional and iteration statements. Finally, for method invocations, we propagate context labels as proposed in [20], whereas they did not implemented it.
- Regarding the confidentiality label inferred for assignment instructions, we improve the granularity of the analysis over previous proposals [2,20] by inferring the confidentiality label during the memory update.
- As a by–product of the verification, a certificate of non-interference 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.

Section 2 introduces the rewriting logic semantics of Java considered in this paper. In Section 3 we present the extended information-flow rewriting logic semantics of Java, and Section 4 formalizes its abstract version. In Section 5 we propose our certification methodology, which we illustrate in Section 6 with some encouraging experimental results that demonstrate the practicality of our approach. Finally, we discuss the related work in Section 7, and Section 8 concludes.

2 The Rewriting Logic Semantics of Java

We assume some basic knowledge of term rewriting [29] and rewriting logic [22]. In the following, we briefly describe the rewriting logic semantics of Java given in [14] and used by the JavaFAN verification tool [15,16]. Its novelty and interest 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 effort[1], even for large languages like Java [15] and the JVM [16]; (iii) the Maude programming language [10], 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 [15].

In [14], 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 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 unity and Δ_{Java} is a set of terminating and confluent (modulo B_{Java}) Σ_{Java}-rewrite rules.

[1] See the different programming languages available at `http://fsl.cs.uiuc.edu/index.php/Rewriting_Logic_Semantics`

Finally, R_{Java} is a set of Σ_{Java}-rewrite rules that are not required to be confluent and terminating. 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 [29,22], we denote by $u \rightarrow^{r}_{\text{Java}} v$ the fact that concrete terms u, v, denoting Java program states, are rewritten (at the top position, see [14]) 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 $u \rightarrow_{\text{Java}} v$ when no confusion can arise. We denote by $\rightarrow^{*}_{\text{Java}}$ the extension of $\rightarrow_{\text{Java}}$ to multiple rewrite steps, i.e., $u \rightarrow^{*}_{\text{Java}} v$ if there exist $u_1, \ldots, u_k$ such that $u \rightarrow_{\text{Java}} u_1 \rightarrow_{\text{Java}} u_2 \cdots u_k \rightarrow_{\text{Java}} v$.

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 are implemented as a (multi)-set union operation that builds up a "soup" of elements.

The rewrite theory $\mathcal{R}_{\text{Java}}$ is defined as terms of a concrete sort State, with the main state attributes (i.e., constructor symbols of the algebraic type State) such as `in`, `out`, `mem`, or `store`. They define an algebraic structure which is parametric w.r.t. 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` constructor symbols 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 that define integer and boolean values. 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. 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 [14].

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 since the smaller the number of rules, the more efficient the verification analysis, because the search space is smaller. According to [14], the Java operational semantics contains about 424 equations and only 7 rules, which considerably saves memory and execution time.

The semantics of Java is defined in a *continuation-based style* [23]. 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. For instance, the Java addition operation on

```
---First evaluate arguments
eq k((E + E') -> K) = k((E, E') -> (+ -> K)) .
---Then, 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 obtain location in store from variable name
eq k(Var -> K) env([Var, Loc] Env) obj(Obj)
    = k(#(Loc) -> K) env([Var, Loc] Env) obj(Obj) .
---Then obtain value stored in such location
rl t(k(#(Loc) -> K) id(I) TC) store(Loc, Value, -1] Store)
=> t(k(Value -> K) id(I) TC) store([Loc,Value,-1] Store) .
```

Fig. 2. Continuation-based equation and rule for variable content retrieval

```
---Obtain variable location while keeping expression in the continuation
eq k((Var = E) -> K) = k(getLocation(Var) -> (=(E) -> K)) .
---Once the location is obtained, evaluate expression keeping location
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 the shared memory
rl t(k([Value -> Loc] -> K) id(I) TC) store([Loc,Value',-1]  ST)
=> t(k(K) id(I) TC) store([Loc, shared(Value), -1] ST) .
```

Fig. 3. Continuation-based equations and rules for Java assignment operator

Java integers is specified[2] 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, `int` is the constructor symbol used to denote a Java integer, and `+` with arity[3] 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.

Another important aspect of the semantics is the use of Java variables. In Figure 2 we show how the contents of a Java variable is retrieved from the store in the Java state. The assignment operator for Java variables is specified in Figure 3. Note that the relative order among assignment and retrieval operations is relevant since multiple threads can try to concurrently assign a value to a

[2] The Maude syntax is almost self-explanatory [10]. The general point is that each syntactic element –e.g. a sort, an operation, an equation, a rule– is declared with an obvious keyword: `sort`, `op`, `eq`, `rl`, etc., ended by a space and a period. We denote variables with uppercase letters whereas lowercase letters denote Maude constructor symbols.

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

```
---Evaluates boolean expression keeping the then and else statements
eq k((if E S else S' fi) -> K) = k(E -> (if(S, S') -> K)) .
eq k(bool(true) -> (if(S, S') -> K)) = k(S -> K) .
eq k(bool(false) -> (if(S, S') -> K)) = k(S' -> K) .
```

Fig. 4. Continuation-based equations for if-then-else statement

```
eq t(k(V -> return -> K) holds(Ll') env(Env')
    fstack( fsi(K', (holds(Ll) env(Env) TC)) Fstack) TC')
= t(k(releaseEnv(Env') -> release(Ll, Ll') -> (V -> K')) holds(Ll)
    env(Env) fstack(Fstack) TC) .
```

Fig. 5. Continuation-based equation for return statement

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.

A relevant aspect of the Java semantics for non-interference is the if-then-else statement, shown in Figure 4. Also important for non-interference is the semantic specification of the Java return statement, shown in Figure 5. The return statement restores the previous environment, the held locks and the local thread state from the function stack, and then updates the continuation to release the method local environment and locks, and to restore them from the stack.

3 An Information-Flow Rewriting Logic Semantics for Java

In this section, we develop an information-flow, extended version of the rewriting logic semantics of Java recalled in Section 2. In order to motivate the new semantics with appropriate Java examples, let us first briefly recall the Java modeling language JML [21].

JML is a behavioral interface specification language that accepts Java built-in operators in order to relieve Java programmers from the encumbrance of learning a language-independent formal specification language like OCL [9]. 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 (in `invariant` clauses) and assert statements (with the `assert` clauses), that express first-order logic statements. As a behavior specification language, JML can also describe how the module will behave when assertions are intermixed with the Java code.

The text of an annotation could be either in one line, after the marker `//@`, or in many lines enclosed between the markers `/*@` and `@*/`. In this paper, we

consider lightweight specifications using the simplest JML clauses for Java methods and type specification of the simplest language level 0 (there are six levels of annotations). We use two method specification clauses, the `ensures` clause to indicate the required confidentiality label expected by the code consumer on the result of a function and the `requires` clause to indicate any precondition (`Low` or `High`) on the confidentiality label of a function input parameter. We use `assert` clauses to indicate the confidentiality label of local variables. The JML specifications written as code annotations are treated like Java comments that are ignored by traditional compilers whereas they are automatically handled by our certification methodology.

The problem of verification and certification of program non-interference using information flow analysis, was first considered in [12]. The flow policy is usually represented by a flow relation between security classes that specify the permissible flows between them. Each storage object (constant, scalar variable, array, or file) is assigned to a security class. This assignment is static and inferred from the declarations in the program. A non-interference policy means that variables have fixed confidentiality levels and that inputs with high confidentiality level do not influence outputs of lower confidentiality level [28,7,30,13]. This means that the values stored in the high confidentiality variables cannot flow to the lower confidentiality variables. It is implicitly assumed that constants appearing in a Java program always have the lowest confidentiality level, i.e., the considered Java program is authorized to access secret data but it does not contain secret data in its code.

A non-interference policy can be represented by a relation $\langle L, \leq \rangle$ and a labeling function $Lab : Var \rightarrow L$, where L is the finite set of confidentiality levels, $\leq$ is a partial order between confidentiality levels, and Var is the finite set of program variables. Usually there are two confidentiality levels, i.e., $Conf = \{\texttt{Low}, \texttt{High}\}$, representing respectively the public non-secret data (low confidentiality) and the secret private one (high confidentiality), so that $\texttt{Low} \leq \texttt{High}$. $\langle Conf, \leq \rangle$ forms a lattice where `Low` is the greatest lower bound or *bottom* ($\bot$), `High` is the least upper bound or *top* ($\top$), and the *join* operator ($\sqcup$) is defined as $\texttt{Low} \sqcup \texttt{Low} = \texttt{Low}$ and, otherwise, $X \sqcup Y = \texttt{High}$. This means that values of `Low` labeled variables can flow to `High` labeled variables, but also that values of `High` labeled variables cannot flow to `Low` labeled variables. The information that flows in a program is either explicit or implicit. An explicit illicit flow is caused by assignment statements in which the values of expressions with high variables are assigned to low variables [28,19], shown in the following.

Example 1. Consider the simple Java program borrowed from [20]. We use the `requires` and `ensures` clauses and the operator `\result`. This example has an illegal direct flow from the variable `high` with confidentiality label `High` to the variable `low` with label `Low` in the first assignment statement. Nevertheless, the final outcome is an integer constant value with the `Low` confidentiality label, so that the final output is legal.

```
public int mE1(int high,int low) { low = high; low = 2; return low;}
/*@ requires high == High && low == Low; @ ensures \result == Low; @*/
```

Another explicit illicit flow might occur in function and procedure invocations, shown in the following example.

Example 2. Consider the following Java program borrowed from [30], whose method `mE522` calls the method `decrementing` with two parameters. The explicit illicit flow occurs at the `decrementing` invocation, which passes the `High` variable `high` to the `Low` parameter `i`.

```
int decrementing(int high,int i)  { high = high - 1; return i; }
/*@ requires high == High && i == Low; @ ensures \result == Low;  @*/
int mE522(int high,int low){ low=decrementing(high,high); return low;}
/*@ requires high == High && low == Low; @ ensures \result == Low; @*/
```

The common source for implicit illicit flows, which can often go unnoticed [28,19], are control flow statements guarded by boolean expressions with variables of confidentiality `High`, shown in the following example.

Example 3. Consider a Java program, also borrowed from [30], with an `if` control flow statement. If the actual data passed to the `low` parameter is not `0` and the returned value is `0`, then we know that the secret variable `high` has a value greater than `2`. Note that the notion of a global confidentiality label (called context label) being updated after each conditional expression is necessary for proper verification of such an implicit leaking [12,20,19].

```
public int mE2(int high,int low) { if (high > 2) low = 0; return low;}
/*@ requires high == High && low == Low; @ ensures \result == Low; @*/
```

In order to avoid false positives, we will dynamically restore the previous global confidentiality label after each conditional construction, as shown in the following example.

Example 4. Consider a slight modification of Example 3 where the returned value does not actually depend on the value of the `High` variable `high`. That is, the variable `j` is affected by the value of the variable `high` but the variable `low` used in the return expression is not.

```
public int mE2*(int high,int low)
/*@ requires high == High && low == Low; @ ensures \result == Low; @*/
{int j=0;low = 0; /*@assert j==Low;@*/ if (high>2) j = 1; return low;}
```

We describe the information-flow extended version of the rewriting logic semantics of Java by the rewrite theory $\mathcal{R}_{\text{Java}^{\text{E}}} = (\Sigma_{\text{Java}^{\text{E}}}, E_{\text{Java}^{\text{E}}}, R_{\text{Java}^{\text{E}}})$, $E_{\text{Java}^{\text{E}}} = \Delta_{\text{Java}^{\text{E}}} \uplus B_{\text{Java}^{\text{E}}}$ and its corresponding $\rightarrow_{\text{Java}^{\text{E}}}$ rewriting relation. In the new semantics, program data do not only consist of standard concrete values but each value is decorated with its corresponding confidentiality label. Our approach consists of extending $\mathcal{R}_{\text{Java}}$ (taking advantage of its modularity) by conveniently complementing the concrete domain Value as to consider the extended domain Value $\times$ LValue.

We introduce the sort LValue to represent values `Low` and `High`. We write `<Value,LValue>` for a pair of a concrete value and its corresponding confidentiality level label. We must also provide appropriate versions of the Java constructions and operators for the new extended domain. Recall that the symbols `env` and `store` are the constructor symbols used by the original Java rewriting semantics for the program environment and the memory store, respectively. The new constructor symbol `lenv` is used to store the global confidentiality level (context label).

Regarding confidentiality, we consider the following Java expressions as a special case of the evaluation: literal constants, variable access, binary operators, assignment expressions, unary pre– and post–fix operators and return expressions. Thanks to the modularity of the rewriting logic approach to formalizing program semantics, our changes to the semantics of Section 2 are incremental and minimal. Variables receive an initial confidentiality level, which is stored in the memory when the variable or parameter is created. Any operation writing a value in a memory location stores, as the confidentiality label for such variable, the join of the confidentiality label of the value to be written and the context level at that moment, as shown in Figure 6. The label of any Integer constant value, shown in Figure 7, is `Low` as expected, since constants are public data. The label of a variable is the confidentiality label of its value in memory and, therefore, the original equations of Figure 2 need no revision.

For the dynamic labeling of the context, the initial context label of any thread is `Low` as usual [12,2,20,19]. Method invocation propagates context label without changes as proposed in [20]. Assignment and expression statements do not change context label. The context label may change only because of conditional control flow statements to control indirect information flow, as shown in Figure 8. The current context label is stored in the continuation using the new `restoreLEnv` continuation operator, which restores the previous context label upon execution; see the last equation of Figure 8. According to [12,2,20,19], the evaluation of boolean expressions returns a confidentiality level associated to the resulting `true` or `false` value and, possibly, a modified context label. We update the context label in order to reflect the confidentiality level returned by the evaluation of the boolean expression, and then the two branches of the

```
rl t(k([< Value,LValue > -> L] -> K) id(I) lenv(LEnv) TC)
   store([L, Value', -1] ST)
=> t(k(K) id(I) lenv(LEnv) TC)
   store([L, shared(< Value,LValue join LEnv >), -1] ST) .
```

Fig. 6. Rule for the extended memory write

```
eq k(i(I) -> K) = k(<int(I),Low> -> K) .
eq k(b(B) -> K) = k(<bool(B),Low> -> K) .
```

Fig. 7. Equations for extended constant evaluation

```
--- First evaluates the boolean expression
--- and keeps the current context label
eq k((if E S else S' fi) -> K) lenv(LEnv)
    = k(E -> if(S, S') -> restoreLEnv(LEnv) -> K) lenv(LEnv) .
--- Then updates the context label
eq k(< bool(true), LValue > -> (if(S, S') -> K)) lenv(LEnv)
    = k(S -> K) lenv(LEnv join LValue) .
eq k(< bool(false), LValue> -> (if(S, S') -> K)) lenv(LEnv)
    = k(S' -> K) lenv(LEnv join LValue) .
--- New equation to restore previous context label
eq k(restoreLEnv(LEnv) -> K) lenv(LEnv') = k(K) lenv(LEnv) .
```

Fig. 8. Continuation-based equations for the extended if-then-else statement

```
eq t(k(< V,LValue > -> return -> K) holds(Ll') env(Env')
      lenv(LEnv) fstack(fsi(K',(holds(Ll) env(Env) TC)) Fstack) TC')
= t(k(releaseEnv(Env') -> release(Ll,Ll') -> (<V,LValue join LEnv> -> K'))
      holds(Ll) env(Env) fstack(Fstack) TC) .
```

Fig. 9. Continuation-based equation for return statement

conditional expression will use such a new context confidentiality label for memory updates.

The extended semantics for the `return` statement considers not only the confidentiality label of the value to be returned but also the context confidentiality level, as shown in Figure 9.

4 The Abstract Rewriting Logic Semantics of Java

In this section, we develop an abstract version of the extended rewriting logic semantics of Java developed in Section 3, described by the rewrite theory $\mathcal{R}_{\text{Java}^\#} = (\Sigma_{\text{Java}^\#}, E_{\text{Java}^\#}, R_{\text{Java}^\#}), E_{\text{Java}^\#} = \Delta_{\text{Java}^\#} \uplus B_{\text{Java}^\#}$ and its corresponding $\rightarrow_{\text{Java}^\#}$ rewriting relation. As in Section 3, our approach for the abstract Java semantics consists of extending the original theory $\mathcal{R}_{\text{Java}}$ (taking advantage of its modularity) by abstracting the domain to $\mathsf{LValue} = \{\mathtt{Low}, \mathtt{High}\}$, and introducing approximate versions of the Java constructions and operators tailored to this domain.

An *abstract interpretation* (or abstraction) [11] 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 [11]. Let $\iota : \alpha(\wp(\mathsf{State})) \to A$ be an isomorphism. Then, given an upper closure operator $\alpha : \wp(\mathsf{State}) \to \wp(\mathsf{State})$, 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 [11] for further details). In our approach [1], we only need an abstract function for each Java variable name `x`, e.g., $\alpha_{\mathtt{x}} : \wp(\mathsf{Value}) \to \wp(\mathsf{Value})$ and homomorphically extend those abstract functions to an abstract function $\alpha : \wp(\mathsf{State}) \to \wp(\mathsf{State})$. Indeed, for each variable `x`, α abstracts the values stored in the Java memory for `x` using $\alpha_{\mathtt{x}}$.

In this section, our abstraction function $\alpha : \wp(\mathsf{State}^E) \to \wp(\mathsf{State}^E)$ is an homomorphism extension to sets of states of the function $2nd : \mathsf{Int} \times \mathsf{LValue} \to \mathsf{LValue}$, meaning that we disregard the actual values of data.

In the abstract Java semantics, several alternative computation steps of $\to_{\text{Java}^{\text{E}}}$ are mimicked by a single abstract computation step of $\to_{\text{Java}^{\#}}$, reflecting the fact that several distinct behaviors are compressed into a single abstract state (i.e. set of states). Consider e.g. the approximate version of the Java $>$ operator. For the case of comparing two abstract states SSt_1 and SSt_2 in $\wp(\mathsf{State}^E)$ for $>$, an (inaccurate) approximation of the result is the set {`<true,Low>`, `<true,High>`, `<false,Low>`, `<false,High>`}, since all combinations are possible when we would compare concrete states. As explained in [1], the instrumentalization of the Java semantics to deal with a set of states instead of one single state implicitly means too many modifications.Therefore, we adopt a different approach. When several $\to_{\text{Java}^{\text{E}}}$ rewrite steps are mimicked by a single abstract rewriting state leading to 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.

Now, we are ready to formalize the abstract rewriting relation $\to_{\text{Java}^{\#}}$, 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 $\alpha(\{l\}) \to \alpha(\{r\})$ by $\alpha(\{l\} \to \{r\})$.

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

$SSt_1 \to_{\text{Java}^{\#}} SSt_2$ *using* $\alpha(\{l\} \to \{r\}) \in (R_{\text{Java}^{\#}} \cup \Delta_{\text{Java}^{\#}})$
iff $\forall u \in \alpha(SSt_1), \exists v \in SSt_2$ *s.t.* $u \to_{\text{Java}^{\text{E}}} v$, *using* $l \to r \in R_{\text{Java}^{\text{E}}} \cup \Delta_{\text{Java}^{\text{E}}}$.

We denote by $\to^*_{\text{Java}^{\#}}$ the extension of $\to_{\text{Java}^{\#}}$ 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 $\alpha : \wp(\mathsf{State}^E) \to \wp(\mathsf{State}^E)$ be an abstraction. Let $SSt_1, SSt_2 \in \wp(\mathsf{State}^E)$. If $SSt_1 \to^*_{\text{Java}^{\#}} SSt_2$, then for all $u \in \alpha(SSt_1)$, there is $v \in SSt_2$ such that $u \to^*_{\text{Java}^{\text{E}}} v$. Let $St_1, St_2 \in \mathsf{State}^E$. If $St_1 \to^*_{\text{Java}^{\text{E}}} St_2$, then there exists $SSt_3 \in \wp(\mathsf{State})$ s.t. $\alpha(St_1) \to^*_{\text{Java}^{\#}} SSt_3$ and $St_2 \in SSt_3$.*

```
rl t(k([LValue -> L] -> K) TC) store([L, Value'] ST)  lenv(LEnv)
=> t(k(K) TC) store([L, LValue join LEnv] ST) lenv(LEnv) .
```

Fig. 10. Abstract rule for the memory write

```
eq k(i(I) -> K) = k(Low -> K) .
```

Fig. 11. Abstract equation for constant evaluation

```
eq t(k(LValue -> return -> K) holds(Ll') env(Env') lenv(LEnv)
      fstack(fsi(K', (holds(Ll) env(Env) TC)) Fstack) TC')
= t(k(releaseEnv(Env') -> release(Ll, Ll') -> LValue join LEnv -> K')
     holds(Ll) env(Env) lenv(LEnv) fstack(Fstack) TC) .
```

Fig. 12. Abstract equation for return statement

Therefore, in the following, we abstract the semantics of Section 3 so that (i) each pair `<Value,LValue>` in the equations and rules are approximated by the second component `LValue`; and (ii) those equations that cannot be proved confluent[4] after the transformation are transformed into rules, to reflect the different possible rewrites denoted by an abstract state. We additionally add a rule to deal with confidentiality values alone, shown in Figure 10. For the label of an integer constant value, we return `Low` as expected, shown in Figure 11. Note that this can be still expressed by means of an equation, since confluence and coherence [10] are preserved. The label of a variable is the confidentiality label of its value in memory and, therefore, we keep the original equations of Figure 2. Analogously, regarding conditionals the equations of Figure 8 still work. Since pairs `<Bool,LValue>` are handled by the return statement, its abstract semantics is still given by the equation of Figure 9.

However, we must add an extra equation to deal with confidentiality values alone, shown in Figure 12, which is almost identical to equations in Figure 9. The following example illustrates the mechanization of the abstract Java semantics.

Example 5. Consider the Java program together with the call to the main function of Example 1. In the search command below, we ask for all possible values returned by the `main` Java function of Example 1.

```
search in PGM-SEMANTICS-ABSTRACT :
 java((preprocess(default class t('Safe1NonInterference) imports nil
 extends Object implements none {(public static) int 'mE1((int d('high)),
 (int d( 'low))) throws( noType) {((10@('low = 'high;)) 1 @('low = i(2);))
 12@ return 'low ;} (public static) void 'main (t('String) [] d('args))
 throws(noType) {5 @ ('System . 'out . 'println < 'mE1 < i(1), i(0)
```

[4] See the Church-Rosser checker for Maude available at `http://www.lcc.uma.es/~duran/CRC/`

```
 > > ;)}}) noType . 'main < new string [i(0)] > noVal))
=>! X:Output .
Solution 1 (state 1)
states: 2  rewrites: 248 in (7ms real) (0 rewrites/second)
X:Output --> pl(Low)
No more solutions.
```

The search command returns that one unique possible abstract Java execution trace is possible, which leads to the abstract value `Low` as the outcome of the Java instruction "`System.out.println(mE1(1,0));`".

5 Certifying Java Source Code

Example 5 above illustrates how our methodology generates a safety certificate which essentially consists of the set of (abstract) rewriting proofs of the form $t_1 \rightarrow^{r_1}_{\text{Java}\#} t_2 \cdots \rightarrow^{r_{k-1}}_{\text{Java}\#} t_k$ that implicitly 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 inexpensively 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 that no rewriting chain has been disregarded, which essentially amounts to use the matching infrastructure available 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 using the equations are omitted.

6 Experiments

The certification methodology presented here has been implemented in Maude[5]. 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 requester, and 3) hide the technical details from the user. The prototype system 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 and with program non-interference.

In Table 1, we study three key points for the practicality of our approach: the size of the reduced certificate versus the Java source code, the size of the reduced certificate versus the size of the full certificate and the relative efficiency of producing certificates w.r.t. their generation. The experiments have been performed on a MacBook with 2 Gb RAM. Programs mE1 and mE2* are Java programs

[5] It is publicly available at
`http://www.dsic.upv.es/users/elp/toolsMaude/rewritingLogic.html`

Table 1. Certificate sizes, and certification times

Code example	Full Cert. Size (Kb)	Red. Cert. Size (Kb)	Size Relation (Red/Source)	Full C. Gen. Time (ms)	Red. Cert. Gen. Time (ms)
mE1	443	2.62	2.29	16	3.5
mE2*	561	4.65	4.97	213.5	28.5
mE2v1E1	615	4.74	4.42	267	47
mE2mE1	578	4.66	3.98	245.5	31
mE522v1	604	2.91	1.67	14	3.5
mE3	553	4.55	4.33	377	57.5

containing the methods of Examples 1, and 4, respectively. Programs mE2mE1 and mE2v1E1 contain methods which are a sequential composition respectively of Examples 3 and 1 and of a variation of Example 3 with Example 1. Program mE522v1 is a non-interferent variation of Example 2. Program mE3, borrowed from [20], is similar to program mE2mE1 using the equality == operator.

The two columns for "Full Cert." show the size in Kbytes and the generation time, respectively, for the full certificates. Similarly for the two columns of "Red. Cert.". Running times are given in milliseconds and were averaged over a sufficient number of iterations. The experiments are very encouraging, since they show that the reduction in size of the certificate is very significant in all cases, ranging the quotient "Red. Cert. Size/Full Cert. Size" from 8.2% in mE2* to 4.8% for mE522v1. When we compare the time employed to generate the full and reduced certificates we have that the reduced certificate generation time takes only 12% of the full certificate generation time.

7 Related Work

Standard Java verification tools that use standard JML [21] as property specification language do not support non–interference certification. Some sophisticated non–interference policies can be expressed by using the JML extensions of the Krakatoa Java verification tool [13]. These JML extensions were developed for Hoare-style assertions regarding program self-composition [4], which means duplicating the code of the program thus requiring to distinguish the same program variables in its two runs. However, non-interference policies that require labeling data variables with confidentiality levels cannot be expressed by using these JML extensions. The confidentiality aspect of non–interference is expressible using the JML specification pattern suggested in [20,30]. Unfortunately, this proposal abuses notation by identifying confidentiality levels with values of the program variables, and it cannot be applied in all cases [30].

Although non–interference has not been considered in current PCC implementations, there are some (not yet implemented) proposals for a subset of Java [5], Java bytecode [27,7,6] and some simple, toy imperative languages [19,8]. However, none of them uses JML to express non–interference policies. [5] proposes a type system, so that a compiler preserving the information flow type could be developed for Java source code. [6] defines the first information flow type system for a sequential JVM-like language that guarantees non–interference in type

checked programs. The soundness was proven by using the theorem prover Coq, and a certified verifier was extracted from the proof. The certified verifier could be used as a PCC proof checker in the consumer's side. Although we consider only two security levels we can easily extend our methodology to the multilevels of confidentiality of [6]. Our global policies attach security levels to object fields but we do not consider heaps (where objects and their fields are stored). Our local policies are very flexible since the security levels of local variables and parameters of methods may change temporarily as in [19,20].

Some proposals also exist for non–interference verification that are based on information flow analysis by using abstract interpretation [2,3,18,17]. However, these proposals do neither generate a proof as a result of the verification nor use JML to express non–interference policies. The idea of first enriching the original semantics of the language by pairing each data value to its security level, and then approximate it by only considering the security level is also in [2]. A similar idea is developed in [17] where input and ouput channels are associated with security levels. Regarding the values delivered by a `return` statement, our work is similar to [3] and [17]. [18] introduces the notion of abstract non–interference: abstract non–interference can be obtained by weakening the standard notion of non–interference by making it parametric relatively to input/output abstractions. In abstract non–interference, the abstract domains encode the allowed flows that characterize the degree of precision of the knowledge of a potential attacker observing the data.

To verify non–interferent Java source programs, there are other type based proposals that do not use JML either to specify information flow policies, namely the Java extensions JFlow [24] and Jif [25]. These compilers produce secure Java source code for verified programs written in the languages JFlow and Jif by first applying static information flow analysis based on type systems to track the correspondence between the confidential information and the policies that restrict its use.

8 Conclusion

As far as we know, we propose the first sound and complete, fully automatic certification technique that applies to certifying non-interference of Java source 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 a sufficiently large subset of the full Java 1.4 language [14]. 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. Our certification methodology extends to other programming languages by simply replacing the concrete semantics by a semantics for the programming language at hand, see [23].

Our work can be easily extended to cope with procedure methods, exceptions, heaps, and multithreading, since they are considered in the Java rewriting logic

semantics. Since we inherit from Maude and the Java rewriting semantics its competitive performance (see [15]), we have a scalable technique that can be further refined to certifying industrial complex Java programs.

References

1. Alba-Castro, M., Alpuente, M., Escobar, S.: Automatic certification of Java source code in rewriting logic. In: Leue, S., Merino, P. (eds.) FMICS 2007. LNCS, vol. 4916, pp. 200–217. Springer, Heidelberg (2008)
2. Barbuti, R., Bernardeschi, C., De Francisco, N.: Abstract interpretation of operational semantics for secure information flow. Information Processing Letters 83(22), 101–108 (2002)
3. Barbuti, R., Bernardeschi, C., De Francisco, N.: Checking security of Java bytecode by abstract interpretation. In: SAC 2002, pp. 229–236. ACM, New York (2002)
4. Barthe, G., D'Argenio, P., Rezk, T.: Secure information flow by self-composition. In: CSFW 2004, pp. 100–114. IEEE, Los Alamitos (2004)
5. Barthe, G., Naumann, D., Rezk, T.: Deriving an information flow checker and certifying compiler for Java. In: SSP 2006, pp. 230–242. IEEE, Los Alamitos (2006)
6. Barthe, G., Pichardie, D., Rezk, T.: A certified lightweight non-interference Java bytecode verifier. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, pp. 125–140. Springer, Heidelberg (2007)
7. Barthe, G., Rezk, T.: Non-interference for a JVM-like language. In: TLDI 2005, pp. 103–112 (2005)
8. Beringer, L., Hofmann, M.: Secure information flow and program logics. In: IEEE CSF 2007, pp. 233–248 (2007)
9. Burdy, L., Cheon, Y., Cok, D., Ernst, M., Kiniry, J., Leavens, G.T., Rustan, K., Leino, M., Poll, E.: An overview of JML tools and applications. IJSTTT 7(3), 212–232 (2005)
10. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.: All About Maude - A High-Performance Logical Framework. LNCS, vol. 4350. Springer, Heidelberg (2007)
11. Cousot, P., Cousot, R.: Systematic Design of Program Analysis Frameworks. In: POPL 1979, pp. 269–282 (1979)
12. Denning, D.E., Denning, P.J.: Certification of programs for secure information flow. Commun. ACM 20(7), 504–513 (1977)
13. Dufay, G., Felty, A., Matwin, S.: Privacy-sensitive information flow with JML. In: Nieuwenhuis, R. (ed.) CADE 2005. LNCS, vol. 3632, pp. 116–130. Springer, Heidelberg (2005)
14. Farzan, A., Chen, F., Meseguer, J., Rosu, G.: JavaRL: The rewriting logic semantics of Java (2007), `http://fsl.cs.uiuc.edu/index.php/Rewriting_Logic_Semantics_of_Java`
15. Farzan, A., Chen, F., Meseguer, J., Rosu, G.: Formal analysis of Java programs in JavaFAN. In: Alur, R., Peled, D.A. (eds.) CAV 2004. LNCS, vol. 3114, pp. 501–505. Springer, Heidelberg (2004)
16. Farzan, A., Meseguer, J., Rosu, G.: Formal JVM code analysis in JavaFAN. In: Rattray, C., Maharaj, S., Shankland, C. (eds.) AMAST 2004. LNCS, vol. 3116, pp. 132–147. Springer, Heidelberg (2004)
17. De Francesco, N., Martini, L.: Instruction-level security typing by abstract interpretation. Int. J. of Inf. Sec. 6(2-3), 85–106 (2007)

18. Giacobazzi, R., Mastroeni, I.: Abstract non-interference: Parameterizing non-interference by abstract interpretation. In: POPL 2004, pp. 186–197 (2004)
19. Hunt, S., Sands, D.: On flow-sensitive security types. In: POPL 2006, pp. 79–90 (2006)
20. Jacobs, B., Pieters, W., Warnier, M.: Statically checking confidentiality via dynamic labels. In: WITS 2005, pp. 50–56 (2005)
21. Leavens, G., Baker, A., Ruby, C.: Preliminary design of JML: A behavioral interface specification language for Java. ACM SIGSOFT Software Engineering Notes 31(3), 1–38 (2006)
22. Meseguer, J.: Conditional rewriting logic as a unified model of concurrency. TCS 96(1), 73–155 (1992)
23. Meseguer, J., Rosu, G.: The rewriting logic semantics project. TCS 373(3), 213–237 (2007)
24. Myers, A.C.: Jflow: Practical mostly-static information flow control. In: POPL 1999, pp. 228–241 (1999)
25. Myers, A.C., Nystrom, N., Zheng, L., Zdancewic, S.: Jif: Java information flow. Software release (2001), `http://www.cs.cornell.edu/jif`
26. Necula, G.C.: Proof carrying code. In: POPL 1997, pp. 106–119 (1997)
27. Rose, E.: Lightweight bytecode verification. J. Autom. Reason. 31(3-4), 303–334 (2003)
28. Sabelfeld, A., Myers, A.C.: Language-based information-flow security. IEEE J. on Selected Areas in Comm. 21(1), 5–19 (2003)
29. TeReSe (ed.): Term Rewriting Systems. Cambridge U. Press, Cambridge (2003)
30. Warnier, M.E.: Language Based Security for Java and JML. PhD thesis, Radboud University Nijmegen (2005)