Formal Proofs of Confidentiality in Java programs

Bart Jacobs and Martijn Warnier

Dept. Comp. Sci., Univ. Nijmegen,
P.O. Box 9010, 6500 GL Nijmegen, The Netherlands
{bart,warnier}@cs.kun.nl

Abstract In this paper we introduce two new ways of establishing confidentiality for sequential Java source code. The first one uses a functional formulation of confidentiality, in what we call *a specification pattern* in the Java Modeling Language JML. It is based on earlier work of Joshi and Leino, and can be proven via standard program verification techniques. Because of limitations in this approach we also consider the relational formulation of confidentiality based on non-interference (going back to Goguen and Meseguer). To support this approach we introduce a novel Hoare logic on relations for proving bisimilarity for Java classes. Both approaches are applied to a non-trivial example: a secure logging mechanism for a cash register. It turns out that the functional approach cannot properly handle information leakage caused by differences in termination modes (such as normal or exceptional), but the relational one can.

1 Introduction

Keeping information secret, or confidential, is one of the more obvious goals of (computer) security. This notion of confidentiality, which originated in work on access control, is probably best known by the approach of Bell and LaPadula [18]. Their model involves so-called mandatory access control and is often described with the requirements "no write down / no read up". Informally one can reduce these to the demands that secret (or high) assets may not leak *any* information to publicly (or low) available sources. In this paper we analyze confidentiality in the context of sequential Java source code. While we only look at Java, the techniques described here can easily be tailored to other imperative or object oriented languages.

Most secure information flow analysis techniques in Java are static, using type-based approaches [3,4,23,29]. One notable exception is [1] which is based on semantics. We shall work in this semantic tradition, and distinguish two approaches, namely a functional and a relational one.

- The functional approach goes back to Joshi and Leino [17]. Our (minor) contribution to it is a reformulation in the (by now standard) Java Modeling Language JML [19]. This reformulation involves a simple pattern of the form `l == \old(--)`, for each low variable `l`. This specification pattern for confidentiality has the advantage that it can be proven with standard verification

techniques, but the disadvantage that it requires a functional description for the values of low variables. Coming up with such a description becomes non-trivial in more complicated situations.
- The relational approach goes back to [12] and relies on the idea of non-interference. It can be expressed in terms of bisimulations. The traditional problem with proving that a particular relation is a bisimulation is that methods must be evaluated twice, once for each x_i in the pairs (x_1, x_2) in the relation. Our (major) contribution here is a new formulation of Hoare logic, this time not with predicates in its triples, but with relations. Via such a logic one only needs to evaluate once in proofs.

The paper starts by explaining bisimulations for classes (in object oriented languages). It then discusses in Section 3 the functional approach to confidentiality for the specification language JML. It is illustrated with several small examples. The Hoare logic for the relational approach appears in Section 4. Both the functional and the relational approach are then applied to a larger example in Section 5. This leads to a comparison and evaluation, showing that the relational approach performs better in the presence of different termination modes. The paper ends with several conclusions and suggestions for future work.

2 Bisimulations and confidentiality

The notion of invariant for a class is well-established in object oriented programming. It is a predicate on the state space that is maintained by all methods, i.e., a subset $P \subseteq \mathsf{State}$ such that for all methods m of the class:

$$\forall x \in \mathsf{State}.\, P(x) \Longrightarrow P(m(x)). \tag{1}$$

Additionally, one should require that P is established by the constructors. Invariants form a fundamental construct in JML; they are typically used to make assumptions explicit that the programmer has in the back of his/her mind.

JML also has the notion of constraint. It is a relation (or binary predicate) between pre- and post-states of method calls. It is thus a relation between states in the same run. We shall see an example of a constraint in Section 5.

A bisimulation is a relation between states of different runs. Bisimulations appeared in the context of transition systems and process calculi, see e.g., [25,22]. They are studied in abstract form in the context of state-based computation provided by the theory of coalgebras [27,13,26]. This forms a source of inspiration for the current work, since the underlying semantics of Java has a coalgebraic formulation (see [15] for a discussion).

Bisimulations for a class are less familiar in object-oriented programming. The basic idea is, like in (1), that they are relations that are maintained by all methods. This can be expressed as: a relation $R \subseteq \mathsf{State} \times \mathsf{State}$ that statisfies for all methods[1] m of the class under consideration:

$$\forall x, y \in \mathsf{State}.\, R(x, y) \Longrightarrow R(m(x), m(y)) \tag{2}$$

[1] We ignore constructors at this stage.

Bisimilarity is then defined as the union of all bisimulations. It is itself a bisimulation—and an equivalence relation—and is the formalisation of the notion of indistinguishability of states. This means that we regard two states x, y as "equal as far as we can see from the outside" in case we can find a bisimulation R with $R(x, y)$.

2.1 Bisimulations for classes

For the relational formulation of a confidentiality property for a class (in an object oriented language) we assume that the fields of the class are partitioned into "high" and "low". We wish to express that the information that is held in the high fields remains confidential. This is done via the notion of non-interference, introduced in [12]. We take as relation $L \subseteq \mathsf{State} \times \mathsf{State}$,

$$L = \{(x, y) \mid \ell(x) = \ell(y), \text{ for all low fields } \ell\}$$

and prove that L is a bisimulation. This bisimulation property expresses confidentiality because if we have two states x, y where we know that low fields are equal, but where we have no information about the high fields, then after each method call these possibly different high values of x and y should not interfere with the low fields: the lows should still be equal. Hence confidentiality says that equality of low fields should be maintained.

Dually, one may formulate integrity as: equality of high fields should be maintained. This is a "no write up", from [6]: differences in low fields should not be visible in the high fields. Since this dual property of integrity is formally the same as confidentiality, we shall not elaborate it in this paper.

2.2 Proving that relations are bisimulations for Java classes

When one actually tries to prove that a particular relation is a bisimulation for a class in a concrete language like Java, it quickly becomes clear that the above formulation (2) is an over-simplification, and that several additional aspects have to be taken into account.

1. Methods in Java have different termination modes: they may hang, terminate normally, or throw an exception[2]. This means that instead of $R(m(x), m(y))$ in (2) we should write: $m(x)$ and $m(y)$ should have the same termination mode, and in that mode the result states (if any) should be related again by R.
2. Parameters of methods need to be included, both for states x and y. So if a method m has parameters $\overrightarrow{a}$ we use quantification to fix their values. So instead of requiring $R(m(x), m(y))$ as conclusion, we should write $\forall \overrightarrow{a}.\, R(m(x)(\overrightarrow{a}), m(y)(\overrightarrow{a}))$.

[2] This only describes the situation from the outside. Inside method bodies we may also have abrupt termination because of a return, break or continue.

3. Methods in Java can also return a result (when they are non-void). We should require in our adaptation of (2) that these results are again indistinguishable. What this means will depend on the type of the result: if the results of $m(x)$ and $m(y)$ are of
   – primitive type, then they should be equal;
   – reference, but no-array, type, then they should be either both null, or both be references to "indistinguishable" objects;
   – array type, they should either be both null, or both be references to arrays with the same length and with "indistinguishable" entries at each position.
4. For each (public) field f of the class, we should have that if $R(x, y)$, then $f(x)$ and $f(y)$ should be indistinguishable, like we just described for results of methods.
5. The definition of (2) requires that each method m is evaluated twice, once in state x and once in state y. This is very inconvenient in proofs. The Hoare logic for relations that we introduce in Section 4 is designed precisely to handle this problem. It uses relations instead of predicates in its triples, and uses structural rules to prove relational program properties. As far as we are aware, such a Hoare logic for relations is new.

Within the Java verification work centered around the LOOP tool [5,15] there is a well-developed semantics for sequential Java. It involves explicit distinctions between the various termination modes in Java, and has therefore no problems with points 1 and 2. Points 3 and 4 are not fully elaborated and tested on examples, but the main ideas are there. Point 5 will be illustrated in this paper, via several proof rules, together with an elaborate example. However, not all proof rules of a relational Hoare logic for Java have been implemented yet for the LOOP tool.

3 The functional formulation of confidentiality in JML

3.1 The Java Modeling Language

The Java Modeling Language, JML [20,19], is a behavioral specification language designed to specify Java Modules. It can be used for classes as a whole, via class invariants and constraints, and for the individual methods of a class, via method specifications consisting of pre-, post- and frame-conditions (assignable clauses). In particular, JML allows within a method specification to describe if a particular exception may occur and which post-condition results in that case.

JML annotations are to be understood as predicates, which should hold for the associated Java code. These annotations are included in the Java source files as special comments indicated by `//@`, or enclosed between `/*@` and `*/`. An example JML method specification of some method `m()` can be seen below:

```
/*@ behavior
  @    requires <precondition>;
```

```
  @  assignable <items that can be modified>;
  @     ensures <normal postcondition>;
  @     signals (E) <exceptional postcondition>;
  @*/
public void m()
```

Such method specifications may be understood as an extension of correctness triples $\{P\}m\{Q\}$ used in Hoare logic, because they allow both normal and exceptional termination. Moreover, the postconditions in JML are relations, because evaluation in the pre-state, indicated by `\old(--)`, may occur. There are a number of different tools available for checking JML annotations ranging form the JML run-time assertion checkers [8] via a static checkers as ESC/JAVA [28,11,10] to theorem proving approaches as Krakatoa [9] Jive [21] and our own tool, the LOOP tool [5,15].

JML specifications are functional specifications. The language is not specifically tailored for specifying security properties like confidentiality or integrity. However if these properties can be expressed in JML the numerous tools available for it can be used to check these properties. In this section we will further explore how one can use JML for specifying confidentiality properties.

3.2 A specification pattern for confidentiality

In [17] Joshi and Leino define a semantical notion of confidentiality as an equality relation ($\doteq$) for a program S in terms of a composition with a special program HH. Assuming we have high variables 'h' and low variables 'l' the program HH is defined as "assign h to an arbitrary low value l". Confidentiality can then be characterized as:

$$\texttt{h} \text{ in } \mathsf{S} \quad \text{is confidential} \equiv (\mathsf{HH} \;\;;\;\; \mathsf{S} \;\;;\;\; \mathsf{HH} \;\doteq\; \mathsf{S} \;\;;\; \mathsf{HH})$$

This can be understood as follows: The two programs are equal provided that the final `l` variables do not depend on an initial value of a `h` variable. Phrased slightly differently we can say that all low variables `l` in the post-state are independent of the value of high variables `h` in the pre-state. Notice that there is a relation between the pre and post state here (in the same run). In JML it is possible to express such relations using the keyword `\old()`. If used in a postcondition, variables encapsulated by `\old` are evaluated in the precondition. This makes it possible to use a reformulation of Joshi and Leino's definition of confidentiality in JML:

```
ensures l == \old(+++)   // with no h in +++
```

So if we can prove for any Java method that there is no low variable `l` that depends on a high variable `h` in the pre-state we have proved confidentiality for that Java method[3]. We will use this `l == \old(+++)` as a *specification pattern*

[3] Of course the specification pattern has to be meaningful, e.g., `l = \old((h==1)? l=0: l=0)` does not express a breach of confidentiality.

for confidentiality. In practice we will often do the reverse and try to prove that some low variable actually depends on some high variable's value in the pre-state. Integrity can easily be expressed using a similar specification pattern: `h == (+++)` where `+++` is again a Java expression that can not contain any low variable `l`.

The group behind the KEY-tool [2] uses a similar approach to prove confidentiality [1]. However they do not use a separate specification language but express confidentiality directly in the dynamic logic which they use to reason about sequential Java programs.

In the next subsection we will look at a simple example.

3.3 Conditionals

Consider the Java Program in Figure 1. Both `h` and `l` are Java `int`'s. Variable `h` represents secret information and `l` public information.

```
/*@   requires true;
  @ assignable l;
  @    ensures l == \old( (h==1) ? 1 : 0 );
  @*/
void test() {
        if (h == 1) l = 1; else l = 0;
}
```

Figure 1. Conditionals can break confidentiality

In the `ensures` clause of the JML specification we use the specification pattern for confidentiality to express that this method leaks information and thereby breaks confidentiality. The value of the variable `l` depends on the old value of `h`. To prove that the specification is indeed correct we use the LOOP-tool to translate the JML specification and Java implementation into the theorem prover PVS [24]. Within PVS we have a semantics of Java & JML so that we can prove the correctness of the specification. Usually this involves some user interaction, typically consisting of applying (ordinary) Hoare logic rules where the user has to come up with appropriate intermediate predicates. This particular example is so trivial that no user interaction is needed to prove the correctness conditions. A more complicated example will we discussed in Section 5. Other examples of JML specifications that have been proved with the LOOP-tool can be found in [7,16].

Type-based approaches like the one developed by Volpano *et. al.* [30] typically find these information flow leaks as well. However if we add one additional line to our program in Figure 1 we obtain the program in Figure 2 with a trivial specification.

```
/*@   requires true;
  @ assignable l;
  @    ensures l == \old(0)
  @*/
void test() {
        if (h == 1) l = 1; else l = 0;
        l = 0;
}
```

Figure 2. Conditionals do not have to break confidentiality

By applying the specification pattern we see that confidentiality stays intact. Type-based approaches consider this program to be flawed. The type rules are too restrictive because they have rules that force statements in the branches of an `if-then-else` to have the same (or a higher) level of classification as the conditional. The semantical approach of Joshi and Leino does not have this restriction. Of course this comes at a cost, we have to do interactive theorem proving to prove confidentiality where type-based approaches can be handled automatically by a type-checking algorithm.

Besides the obvious advantage of having a range of tools to check JML specifications there are also two more negative points associated with this approach:

- In essence the specifier has to know *in advance* where confidentiality will be violated. Once this is known, proving it should be relatively straightforward.
- These functional description can become very complex for larger programs. In practice we think this approach is only feasible for small programs with only a few variables.

4 A Hoare logic on relations for confidentiality

Classical Hoare logic allows one to reason about simple imperative languages which contain assignments, conditional expressions, while and for loops and block statements. Such a logic involves triples $\{P\}m\{Q\}$ where P and Q are predicates. An extended version of such a Hoare logic for Java is available with the LOOP tool (see [14]) and has been used in many applications.

4.1 A Hoare Logic on Relations

As explained in Section 2 the relational approach to confidentiality requires proofs that specific relations are bisimulations, i.e., are maintained by all methods. In this section we introduce a Hoare logic on relations for a simple imperative language. We will use small case letters p, q, r as expressions in our language and capital letters R, T, S are relations. For a predicate c we use a short hand notation c^2 ("c squared") for the relation given by $c^2(x, y) = c(x) \wedge c(y)$.

The main rules for our logic are then as follows.

$$\frac{S \Rightarrow R[a/v(x), a/v(y)]}{\{S\} v := a \{R\}} \text{ assignment} \tag{3}$$

$$\frac{\{R\}\ p\ \{T\} \qquad \{T\}\ q\ \{S\}}{\{R\}\ p; q\ \{S\}} \text{ composition} \tag{4}$$

$$\frac{R \Rightarrow c^2 \vee (\neg c)^2 \qquad \{R \wedge c^2\}\ p\ \{S\} \qquad \{R \wedge (\neg c)^2\}\ q\ \{S\}}{\{R\}\ \text{if } c \text{ then } p \text{ else } q\ \{S\}} \text{ if-then-else} \tag{5}$$

$$\frac{R \Rightarrow c^2 \vee (\neg c)^2 \qquad \{R \wedge c^2\}\ p\ \{S\}}{\{R\}\ \text{while } c \text{ do } p\ \{(\neg c)^2 \wedge S\}} \text{ while (partial)} \tag{6}$$

$$\frac{R \Rightarrow c^2 \vee (\neg c)^2 \qquad [R \wedge c^2 \wedge (\text{variant} = n)^2]\ p\ [S \wedge (\text{variant} < n)^2]}{[R]\ \text{while } c \text{ do } p\ [(\neg c)^2 \wedge S]} \text{ while (total)} \tag{7}$$

Notice that we do not cover all possible cases in our rules. For instance, in proof rule (5) we use a conditional c^2. Formally there are four different cases possible: $c(x) \wedge c(y)$, $\neg c(x) \wedge \neg c(y)$, $\neg c(x) \wedge c(y)$ and $c(x) \wedge \neg c(y)$. We exclude the latter two cases because we do not need them for proving confidentiality. If the boolean values of $c(x)$ and $c(y)$ are different this will imply that confidentiality is broken at this point in the program. The same also applies to the proof rules (6,7). We simply duplicate the variant in the total correctness rule (7). A variant gives a mapping from the underlying state space to some well-founded set. Termination can then be proved by showing that for each execution of the body of the while-loop this mapping decreases. Formally we have to define two mappings, but for our approach one mapping suffices. Again if the variant is different for separate states confidentiality will be broken. We will now show how we use the bisimulations from Section 2 in combination with the Hoare logic for relations introduced here for reasoning about confidentiality in Java programs. The programs from Figure 1 and 2 from Section 3.3 will again be used to illustrate this.

4.2 Conditionals revisited

We apply our Hoare logic on relations to prove (a breach of) confidentiality for the programs from Figures 1 and 2. Since our notion of confidentiality is a class-wide property we assume that both methods are declared in their own class each having only two fields, namely `h` and `l` both `int`'s. Proving that the programs maintain confidentiality in this case means that we have to prove that the relation $\{(x, y) \mid \mathtt{l}(x) = \mathtt{l}(y)\}$ is a bisimulation.

We assume that this relation holds in the pre-state and try to prove that it also holds in the post-state. We start with applying rule (5) from Section 4.1. Here it immediately becomes clear that we can not prove that in the post-condition $\texttt{l}(x) = \texttt{l}(y)$ holds. Since we have no information about variable `h`, different values of `h` in state x and y implies different values for `l` in states x and y, thereby breaking confidentiality.

The program from Figure 2 is done as follows. We again want to prove that the relation $\texttt{l}(x) = \texttt{l}(y)$ is a bisimulation. Assuming that this relation holds in the pre-state we apply the composition rule (4) from our Hoare logic on relations. As an intermediate relation—the T in rule (4)—we can simply take `true`. The proof goal on the left in the composition rule then trivially holds. And the one on the right is $\{\texttt{true}\}\texttt{l=0}\{\texttt{l}(x) = \texttt{l}(y)\}$. It follows from the assignment rule. This shows that the relation $\{(x, y) \mid \texttt{l}(x) = \texttt{l}(y)\}$ is a bisimulation, and thus that confidentiality is preserved.

5 An example: logging a Cash Register

Most modern cash registers have some sort of logging mechanism to make it possible to check that no money from the register has disappeared. We have written a small Java program that implements such a secure logging mechanism. The idea is that all transaction are stored in a `int` array in such a way that both the total amount in the register and each individual transaction is logged. The first index of the array will be set to an initial value and each consecutive index will contain the value of the previous index plus the new amount added to the register. When the last index is used the register will be locked and no additional money can be added to the registers until the log is dumped and the process can continue again. Figure 3 gives a graphical representation of the log.

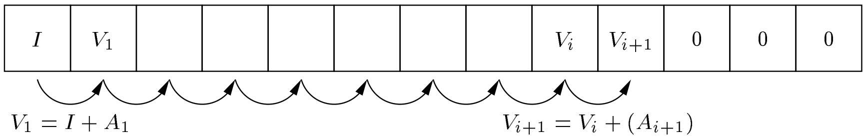

Figure 3. The log of the cash register

Indices that are not used yet have the default value zero. Since all values depend on the previous one, fraud with the log involves changing all values of the log. Moreover the initial value (I) at the first index ensures that someone who can read one index has no idea what to total amount of money in the cash is since she also has to know this initial value. We will look at one method from this logging mechanism, namely the method that allows one to add money to the register. There are two parameters: `amount` which is the amount that has to be payed by the customer and `money` which is the amount of money given by the customer.

The return value is the change that has to be returned to the customer. We will first show the Java implementation and JML specification below and then explain the JML specification.

```
class Logger{

     public int counter, size;
     private /*@ spec_public @*/ int[] payments;

    /*@ invariant
      @   size > 0 && counter >= 0 && counter <= size &&
      @   payments != null && payments.length == size &&
      @   (\forall int i; i > counter && i < size ==> payments[i] == 0);
      @*/

    /*@ constraint
      @   size == \old(size) && payments == \old(payments);
      @*/

    /*@ behavior
      @   requires   counter > 0;
      @ assignable   counter, payments[counter];
      @    ensures   amount > 0 && \old(counter) < size &&
      @              counter == \old(counter + 1) &&
      @              payments[\old(counter)] ==
      @                  \old(payments[counter - 1] + amount) &&
      @              \result == \old(money) - \old(amount) &&
      @              \result >= 0;
      @  signals(NotEnoughMoneyException)
      @              \old(amount) > \old(money) &&
      @              payments[counter] ==
      @                  \old(payments[counter]) &&
      @              counter == \old(counter);
      @  signals(LogFullException)
      @              (\old(counter) >= \old(size)  ||
      @               amount <= 0) &&
      @              counter == \old(counter) &&
      @              payments[counter] ==
      @                  \old(payments[counter]);
      @  signals(OverflowException)
      @              \old(payments[counter - 1]) >
      @              (Integer.MAX_VALUE - \old(amount)) &&
      @              counter == \old(counter) &&
      @              payments[counter] ==
      @                  \old(payments[counter]);
      @*/
    public int add(int amount, int money) throws NotEnoughMoneyException,
                                                 LogFullException,
                                                 OverflowException{
        if (amount > money)
```

```
            NotEnoughMoneyException.throwIt();
         if (amount > 0 && counter < size ) {
            if(payments[counter-1] > Integer.MAX_VALUE - amount)
                OverflowException.throwIt();
            payments[counter] =
                payments[counter-1] + amount;
            counter++;
         }
         else LogFullException.throwIt();
        return money - amount;
    }
}
```

There are three fields associated with this class, a public `counter` that keeps track of where the current index in the private array (the log) `payments` is and a public field `size` that equals the length of this array. The specification has a class invariant which states some obvious properties of the array as well as asserting that indices that are not used yet have the default value zero. The constraint gives a relation between the pre- and post-state and says that once created the length and the reference pointer of the array stay constant.

The method specification is a bit more complicated. First, notice that besides normal termination the method can also terminate with three different exceptions, each resulting in a different post-condition. In the `ensures` clause we specify what should happen after normal termination. The `counter` should be incremented by one, the return value of the method (indicated by the JML keyword `\result`) should be the change for the customer, and the value at the index in the array at place `\old(counter)` should have the value at `(\old(counter)-1)` plus the `amount`. All other values of the array are unmodified as can be seen in the `assignable` clause.

Three exceptions[4] can be thrown: A `NotEnoughMoneyException` occurs if the `money` given by the customer is less than the `amount` that has to be paid. An `OverflowException` will be thrown if the value stored at the current index is bigger than `Integer.MAX_VALUE`, and a `LogFullException` will be thrown if the log is full i.e., if `\old(counter)==size` (or if `amount` is below 0 which shouldn't be possible). The post-conditions for each exception can be found in the corresponding `signals` clauses. Both the `NotEnoughMoneyException` and the `LogFullException` are not that interesting.

The public fields `count` and `size` will be treated as a 'low' variable and the `private` field including its array entries, is considered to be a 'high' or secret variable. When the `OverflowException` is thrown the method will leak information: part of this post-condition reads `\old(payments[counter-1]) > (Integer.MAX_VALUE - \old(amount))` which leaks partial information about the value stored at the previous index (i.e., at `\old(payments[counter-1])`).

[4] As an aside, the exceptions are all thrown implicitly by a static `throwIt()` method. This ensures that only one static instance of each exception exists in stead of creating a new exception every time.

The above specification has been proven for the given implementation, using the LOOP tool. But that is not our main focus. In the next sections we will show that this method indeed leaks information.

5.1 JML specification pattern for confidentiality

This example illustrates that the specification pattern for confidentiality need not be sufficient to guarantee confidentiality. The reason is that we have different termination modes. In each of these we have an equation of the form `counter == \old(+++)`. But clearly, the fact that a specific exception—for overflow—occurs gives away information.

As this example shows, the specification pattern for confidentiality can only be applied when a difference in the value of high variables does not mean that sometimes the method terminates normally and sometimes with an exception (or does not terminate at all). The other problems with this approach still apply: one has to know if and where confidentiality is broken in advance and the functional specifications can become very complex. Notice that all these points also apply to the specification pattern for integrity.

5.2 Relational Hoare logic

We start our proof sketch by defining an appropriate intended bisimulation S. It should include equalities between all public variables, which in this case are `counter` and `size`—for which we shall just write `c` and `s` for convenience.

$$\mathsf{S} = \{(x, y) \mid \mathtt{c}(x) = \mathtt{c}(y) \wedge \mathtt{s}(x) = \mathtt{s}(y)\}$$

Assuming this relation holds for the pre-states we start our proof by applying the composition rule (4) using S as an intermediate relation. In the left branch, the first `if` is then trivial since we immediately exit the method with an exception. We continue with another composition in the right branch, again using relation S as intermediate relation. On the left, by application of rule (5) we enter the big `if-then-else`. Since both `size` and `amount` (as parameter) have the same values for the current states we either continue the proof and encounter the next `if` or we end up in the `else` part which immediately ends the execution of the method with an exception, thereby proving that relation S still holds. We are now at the point where we find the third `if` of the method body, namely:

```
if(payments[counter-1] > Integer.MAX_VALUE - amount)
```

At this stage we do not know what the value of `payments[counter-1]` is: it has a 'high' or 'secret' value, and is not included in our relation. Hence we can not continue with our proof, because different values of the high variables in this case can lead to different termination modes of the method, thereby breaking confidentiality. The fact that our proof comes to halt indicates a breach of confidentiality.

As an aside, we remark that if we would know that both states lead to normal termination—which is not guaranteed—then the relation S is maintained.

This example illustrates that by simply applying the (syntax-driven) rules from our Hoare logic on relations we find the place where confidentiality is broken. The proof presented here is of course not a completely formal proof since we currently only have rules for a simple imperative language—and not for full (sequential) Java.

6 Conclusions and Future Work

In this paper we have explored two distinct ways of proving confidentiality for sequential Java source code: a functional and a relational one.

Expressing the functional notion of confidentiality as proposed by Joshi and Leino via a specification pattern in JML makes it possible to formulate and prove confidentiality and integrity. The number of different tools for checking JML specifications makes it attractive to use this method. However this method also has a number of disadvantages.

- It stays quite *ad hoc*, since one has to know in advance where confidentiality fails.
- It does not work with different termination modes, esp. when these may be caused by differences in the values of high variables.
- For larger examples it may become hard to come up with functional expressions in JML which describing (failure of) confidentiality.

We expect that in the long run using the relational Hoare logic to prove (non-)existence of a class-wide bisimulation expressing confidentiality is more promising. The obvious advantage in this approach is of course that one finds the points where confidentiality fails by simply applying the syntax-driven rules. Much like is the case with normal Hoare logic for program correctness. We believe that eventually it should be possible to find all the points where confidentiality fails. Not only for explicit leaks (as described in this paper, i.e., explicit assignments from high to low variables) but also for certain implicit leaks. The latter are often called *covert channels* and involve among others non-termination behavior (which can leak one bit at a time). Since we want to integrate our new approach in the LOOP tool we imagine that it might make sense to add appropriate new keywords to JML for confidentiality (and integrity), resulting in suitable bisimulation proof obligations.

We are not yet at this point, but for future work we want to elaborate the relational Hoare logic, formalize it in PVS, and integrate it with the existing Hoare logic for predicates. This should involve appropriate, provably sound rules for all the constructs of sequential Java.

Acknowledgments

Thanks are due to Erik Poll for helpful discussions.

References

1. Ádám Darvas, Reiner Hähnle, and Dave Sands. A theorem proving approach to analysis of secure information flow. In Roberto Gorrieri, editor, *Workshop on Issues in the Theory of Security, WITS*. IFIP WG 1.7, ACM SIGPLAN and GI FoMSESS, 2003.
2. W. Ahrendt, Th. Baar, B. Beckert, R. Bubel, M. Giese, R. Hähnle, W. Menzel, W. Mostowski, A. Roth, S. Schlager, and P.H. Schmitt. The KeY tool. Technical Report 2003-5, Chalmers and Göteborg University, 2003. `http://i12www.ira.uka.de/~key`.
3. Anindya Banerjee and David A. Naumann. Stack-Based Access Control for Secure Information Flow, 2003. manuscript.
4. Giles Barthe, Amitabh Basu, and Tamara Rezk. Security Types Preserving Compilation, manuscript, September 2003. available at: `http://www-sop.inria.fr/lemme/Tamara.Rezk/publications.html`.
5. J. van den Berg and B. Jacobs. The LOOP compiler for Java and JML. In T. Margaria and W. Yi, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, number 2031 in Lect. Notes Comp. Sci., pages 299–312. Springer, Berlin, 2001.
6. K. J. Biba. Integrity considerations for secure computer systems. Technical Report MTR-3153, MITRE Corp., 1977.
7. C.-B. Breunesse, J. van den Berg, and B. Jacobs. Specifying and verifying a decimal representation in Java for smart cards. In H. Kirchner and C. Ringeissen, editors, *Algebraic Methodology and Software Technology*, number 2422 in Lect. Notes Comp. Sci., pages 304–318. Springer, Berlin, 2002.
8. Y. Cheon and G.T. Leavens. A runtime assertion checker for the Java Modeling Language (JML). In H.R. Arabnia and Y. Mun, editors, *International Conference on Software Engineering Research and Practice (SERP '02)*, pages 322–328. CSREA Press, Las Vegas, 2002.
9. E. Contejean, J. Duprat, J.-C. Filliâtre, C. Marché, C. Paulin-Mohring, and X. Urbain. The Krakatoa tool for JML/Java program certification. *Journ. of Logic and Algebraic Programming*, to appear. Available via the Krakatoa home page at `www.lri.fr/~marche/krakatoa/`.
10. ESC/JAVA2 project. available at: `http://www.cs.kun.nl/ita/research/projects/loop/`.
11. C. Flanagan, K.R.M. Leino, M. Lillibridge, G. Nelson, J.B. Saxe, and R. Stata. Extended static checking for Java. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, volume 37(5) of *SIGPLAN Notices*, pages 234–245. ACM, 2002.
12. J.A Goguen and J. Meseguer. Security policies and security models. pages 11–20, 1982.
13. B. Jacobs. Exercises in coalgebraic specification. In R. Crole R. Backhouse and J. Gibbons, editors, *Algebraic and Coalgebraic Methods in the Mathematics of Program Construction*, number 2297 in Lect. Notes Comp. Sci., pages 237–280. Springer, Berlin, 2002.
14. B. Jacobs and E. Poll. A logic for the Java Modeling Language JML. In H. Hussmann, editor, *Fundamental Approaches to Software Engineering (FASE)*, number 2029 in Lect. Notes Comp. Sci., pages 284–299. Springer, Berlin, 2001.
15. B. Jacobs and E. Poll. Java program verification at nijmegen: Developments and perspective. Techn. Rep. NIII-R0318, Comput. Sci. Inst., Univ. of Nijmegen., 2003.

16. Bart Jacobs, Joseph Kiniry, and Martijn Warnier. Java Program Verification Challenges. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem-Paul de Roever, editors, *Formal Methods for Components and Objects*, volume 2852 of *Lect. Notes Comp. Sci.*, pages 202–219. Springer, Berlin, 2003.
17. R. Joshi and K.R.M. Leino. A semantic approach to secure information flow. *Science of Comput. Progr.*, 37(1-3):113–138, 2000.
18. L. J. LaPadula and D. E. Bell. Secure Computer systems: A mathematical model. Technical Report MTR-2547, Vol 2, MITRE Corp., 1973. Reprinted in *J. of Computer Security*, vol 4, no 2–3, pp 239–263, 1996 .
19. G.T. Leavens, A.L. Baker, and C. Ruby. JML: A notation for detailed design. In H. Kilov and B. Rumpe, editors, *Behavioral Specifications of Business and Systems*, pages 175–188. Kluwer, 1999.
20. G.T. Leavens, A.L. Baker, and C. Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Techn. Rep. 98-06, Dep. of Comp. Sci., Iowa State Univ.
`http://www.cs.iastate.edu/~leavens/JML.html`, 1999.
21. J. Meyer, P. Müller, and A. Poetzsch-Heffter. The JIVE system—implementation description, 2000. Available at `http://www.informatik.fernuni-hagen.de/pi5/publications.html`.
22. R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.
23. Andrew C. Myers. JFlow: Practical Mostly-Static Information Flow Control. In *Proceedings of the 26th ACM Symposium on Principles of Programming Languages (POPL'99)*, 1999.
24. S. Owre, J.M. Rushby, N. Shankar, and F. von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Trans. on Softw. Eng.*, 21(2):107–125, 1995.
25. D.M.R. Park. Concurrency and automata on infinite sequences. In P. Deussen, editor, *Proceedings 5th GI Conference on Theoretical Computer Science*, number 104 in Lect. Notes Comp. Sci., pages 15–32. Springer, Berlin, 1981.
26. J. Rothe, H. Tews, and B. Jacobs. The coalgebraic class specification language CCSL. *Journ. of Universal Comp. Sci.*, 7(2), 2001.
27. J. Rutten. Universal coalgebra: a theory of systems. *Theor. Comp. Sci.*, 249:3–80, 2000.
28. Extended static checker ESC/Java. Compaq System Research Center.
`www.research.compaq.com/SRC/esc/Esc.html`.
29. Martin Strecker. Formal analysis of an information flow type system for MicroJava (extended version). Technical report, Technische Universität München, July 2003.
30. Dennis Volpano, Geoffrey Smith, and Cynthia Irvine. A sound type system for secure flow analysis. *Journal of computer security*, 4(3):167–187, 1996.