Non-interference in JML

Martijn Warnier and Martijn Oostdijk

Institute for Computing and Information Sciences,
Radboud University Nijmegen,
P.O. Box 9010, 6500 GL Nijmegen, The Netherlands
{warnier,martijno}@cs.ru.nl

Abstract

This paper deals with the specification of non-interference properties in the behavioral specification language JML. The notion of a specification pattern for JML is introduced and it is shown how such patterns can be used to specify non-interference properties such as confidentiality and integrity. The main contribution of this paper is an algorithm that takes a Java source file as input and generates a source file annotated with specification patterns for confidentiality in JML. The algorithm works for input programs written in a non-trivial subset of sequential Java. We prove that the specifications generated by the algorithm are correct and express confidentiality. In case of loops, the specifications will have to be over-approximations. Fortunately, the generated specifications can easily be refined and combined with existing hand-written JML specifications. The resulting specifications can be verified using any of the JML tools.

Keywords: Specification Generation, Secure Information Flow, Language-based Security

1 Introduction

The focus of this paper is the specification of the confidentiality property in the Java Modeling Language (JML [22, 19]), a behavioral specification language for Java. Confidentiality is an important security property that is notoriously hard to enforce in computer programs. Informally, a program is called confidential if no illegal information flow from secret input to public output variables exists. The technical notion of non-interference [15] is used to formalize confidentiality.

Confidentiality can be expressed in JML using the notion of a so-called *specification pattern* for JML. As far as we know, this formalization in JML is novel. Other work has either extended JML to be able to specify confidentiality [10] or formalized it directly in the low-level semantics of a particular tool [9]. Both these approaches lose the ability to use the whole range of JML tools [6] for verification, which is one of *the* attractive features of JML.

Writing JML specifications for Java programs is a tedious and error-prone task. To facilitate this problem a *specification generation* algorithm for a non-trivial subset of sequential Java is introduced. It produces, from a Java source file, a Java program annotated with JML specifications that express confidentiality.

The generated specifications are sound, in the sense that programs that are confidential according to the generated specifications are indeed confidential. The approach is not complete because our treatment of loops is an over-approximation. However, it is easy to refine the over-approximated parts of the generated specifications manually. Another attractive feature of our approach is that we can use

existing JML specifications to generate more precise specifications that express confidentiality.
Organization of the paper: Section 2 starts with a brief introduction to JML, Section 3 introduces the notion of a specification pattern (in JML) and shows how to use this notion to express a JML specification pattern. Section 4 then explains how such patterns can be generated from a Java source file. In Section 5 we formalize the notion of non-interference and state soundness of the specification generation. Two examples are given in Section 6. Section 7 explores how the specify termination sensitive non-interference in JML and Section 8 discusses related work. We end the paper with conclusions and suggestions for future work.

2 The Java Modeling Language

The Java Modeling Language (JML) is a behavioral interface specification language designed for the specification of Java classes. It can be used for classes as a whole, via class invariants and constraints, and for individual methods of a class, via method specifications consisting of pre-, post- and frame-conditions. In particular, it is possible within a method specification to indicate whether a particular exception could occur and what postcondition holds in that case.

JML annotations are to be understood as predicates that may or may not hold for the associated Java code. Such annotations are included in the Java source files as special comments indicated by //@, or enclosed between /∗@ and ∗/. They are ignored by the Java compiler and recognized by special tools such as the JML runtime checker, ESC/Java2, the LOOP tool, the Krakatoa verification condition generator and the JACK tool. An overview of JML tools can be found in [6]. Example 1 shows a JML method specification of some method `m`():

Example 1.

```
/*@ behavior
      requires precondition;
    assignable items that can be modified;
       ensures normal postcondition;
    signals (E) exceptional postcondition
                because of exception E;
 */
public void m()
```

Such method specifications may be understood as an extension of the classical correctness triples $\{P\}m\{Q\}$ used in Hoare logic, because they allow both normal and exceptional termination. More examples of JML specifications are given in Section 6.

JML is intended to be usable by Java programmers. Its syntax is therefore similar to Java's. However, it has a few additional keywords, such as `==>` (for implication), `\old` (for evaluation in the pre-state), `\result` (for the return value of a method, if any), and `\forall` and `\exists` (for quantification).

JML also supports so-called `model` fields [5]. A model field is a field that is not accessible by the Java code, i.e., a specification only field. It should be thought of as the abstraction of one or more concrete fields [7]. Model fields are related to concrete fields via JML's `depends` clause, which indicates the concrete fields on which a model field depends, and a `represents` clause, which specifies exactly how a model field is related to concrete fields.

The `ghost` keyword indicates another specification-only field in JML. Values of ghost fields are not determined by `represents` clauses, instead they are assigned directly using the JML keyword `set`. Values of ghost fields can be changed *inside* the body of a method.

3 A specification pattern for confidentiality

In the examples in this paper we use the simple security lattice Σ, formally it is defined as $\{\mathsf{High}, \mathsf{Low}\}$ with $\mathsf{Low} \sqsubseteq \mathsf{High}$. A secure information flow policy is then given by the function $\mathsf{Sif} : \mathsf{Var} \to \Sigma$ which maps variables to security levels in the simple security lattice. We will abuse notation by identifying security levels and the sets of variables corresponding to those levels, i.e. $\mathsf{High} = \{v \in \mathsf{Var} \mid \mathsf{Sif}(v) \sqsupseteq \mathsf{High}\}$ and $\mathsf{Low} = \mathsf{Var} \setminus \mathsf{High}$.

We formalize confidentiality using the notion of non-interference [15]. Our formalization of non-interference in JML is inspired by the work of Joshi and Leino [20]. They define a semantical notion of confidentiality as an equality relation on fields ($\doteq$) for a program S in terms of a composition with a special program HH. I.e., a program S is confidential if:

$$\mathsf{HH}\ ;\ \mathsf{S}\ ;\ \mathsf{HH}\ \doteq\ \mathsf{S}\ ;\ \mathsf{HH}$$

Where the program HH is defined as "assign arbitrary values to variables of security type High". We will make this definition formal in Definition 8.

For an observer who can only see fields of type Low the two programs are observationally equivalent provided that the final values of Low fields do not depend on initial values of High fields. In other words, low fields `low` in the post-state are independent of the values of the high variables `high` in the pre-state.

Notice that there is a relation between the pre- and post-state here. 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 formulation of confidentiality in JML that is equivalent to Joshi and Leino's definition. This equivalence is made precise in Section 5.

Pattern 1 (specification pattern for confidentiality)**.**

The semantic notion of confidentiality above (also see Definition 8) can be expressed in JML as a specification pattern:

$$\texttt{ensures low} == \texttt{\textbackslash old}(\chi)$$

For all `low` $\in \mathsf{Low}$*. With the restrictions that none of the fields* `high` $\in \mathsf{High}$ *appear inside* χ*.*

By proving for a Java method that all fields *low* $\in \mathsf{Low}$ are independent of fields *high* $\in \mathsf{High}$ in the pre-state, we have proved confidentiality for that Java method[1]. The meta expression `low` $==$ `\old`(χ) is called a *specification pattern* for confidentiality.

Integrity, as the formal dual of confidentiality, can easily be expressed using a similar specification pattern: `high` $==$ `\old`(χ) where χ is again a Java expression that cannot contain any fields `low` $\in \mathsf{Low}$. This paper focuses on proving confidentiality, but all techniques described in the sequel are equally applicable to integrity.

[1] Of course the specification pattern has to be meaningful, e.g., `low` $==$ `\old`$(($`high` $== 1)?$`low` $= 0 :$ `low` $= 0)$ does not prove a breach of confidentiality since this is equivalent to a simpler pattern `low` $==$ `\old`(0) for which it is clear that it does not break confidentiality.

The specification pattern for confidentiality can be used to prove one of the most common –and weakest– forms of non-interference know as *termination insensitive* non-interference. Informally it can be understood to mean that the non-interference property is only specified if the program terminates normally. If the program terminates with an exception or does not terminate at all (loops) a non-interference property is not guaranteed.

Stronger forms of non-interference also take termination behavior into account and even consider so-called covert channels [21], such as timing [1, 3], resource consumption or caching, to leak information. We will not discuss covert channels here, termination behavior is discussed further in Section 7.

4 JML specification generation

This section shows how the specification pattern for confidentiality in JML can be generated from a Java source file. We will not show this for the complete sequential part of Java, but only for a small subset thereof, which we shall call Core Java. We think that this subset is complex enough to show the applicability of our approach. The language supports side effects, both normal and a form of abrupt termination (via return statements), (non-recursive) method calls as well as the usual control flow mechanisms such as branching and looping.

Definition 1 (Core Java)**.** *The syntax for the core Java language is given by*

$$\begin{array}{lrcl} \mathsf{Expr} & e & ::= & c \mid v \mid e_1 \ \mathtt{op}\ e_2 \mid e_1\ ?\ e_2\ :\ e_3 \mid m(\overrightarrow{e}) \\ \mathsf{Statement} & s & ::= & \mathtt{skip} \mid v\ := e \mid s_1; s_2 \mid \mathtt{if}\ (e)\ s_1\ \mathtt{else}\ s_2 \\ & & & \mathtt{while}\ (e)\ s \mid \mathtt{return}\ e \end{array}$$

where op *is either a primitive operator* `+`, `-`, `*`*, a comparison operator* $<, \leq, =$*, or one of the (unconditional) boolean operators* `|` *and* `&`*;* $:=$ *is assignment,* v *ranges over variables,* c *ranges over constants.*

The specification generation algorithm is given by a dedicated strongest postcondition calculus (sp) which takes as argument a list of variables –called an *abstract state list*– and a statement and calculates for each of these how they relate to other fields and parameters in the pre-state. We introduce the following notational conventions: $[\alpha, \beta, \gamma]$ is a list with elements α, β and γ, $[h : t]$ is a list with head-element h and tail t, a single letter l also represents a list.

Two different kind of specification generation rules can be distinguished: (i) precise rules for the non-looping part of Core Java. These give an exact specification in JML of the specification pattern for confidentiality, and (ii) approximate rules for complete Core Java, these give an over-approximation that manually can be refined by the user.

4.1 Specification generation for the non-looping statements

For presentation purposes we first explain the basic idea for the subset of Core Java that only contains skip, assignment, composition and if-then-else as possible statements and does not allow side-effects in expressions, nor return statements, method calls or loops. Definition 2 should explain the basic idea.

Definition 2 (Rules for non-looping statements)**.**

$$\begin{array}{lcl} \mathsf{sp}(l, \mathtt{skip}) & = & l \\ \mathsf{sp}(l, x := e) & = & l\langle x \mapsto \mathsf{val}(e, l)\rangle \\ \mathsf{sp}(l, s_1; s_2) & = & \mathsf{sp}(\mathsf{sp}(l, s_1), s_2) \\ \mathsf{sp}(l, \mathtt{if}\ (e)\ s_1\ \mathtt{else}\ s_2) & = & \end{array}$$

$$
\begin{array}{ll}
\texttt{let} & l_1 = \mathsf{sp}(l, s_1) \\
& l_2 = \mathsf{sp}(l, s_2) \\
\texttt{in} & [x_i \mapsto \left\{ \begin{array}{ll} l_1[x_i] & \mathit{if}\ l_1[x_i] = l_2[x_i] \\ e\ ?\ l_1[x_i]\ :\ l_2[x_i] & \mathit{otherwise} \end{array} \right. \mid i \in \mathbb{N}]
\end{array}
$$

The function sp above calculates for each variable in scope how it relates in the post-state to variables in the pre-state of a method. Each variable in the state list is coupled with an expression. In the pre-state each variable is coupled with itself and when an expression is assigned to it the expression is coupled with the assigned variable. This can be seen in the assignment rule. We use the val function, defined below, to relate the variable x to an expression that is to be evaluated in the pre-state.

Definition 3. *We define* $\mathsf{val} : \mathsf{Expr} \times \mathsf{List} \to \mathsf{Expr}$ *inductively as follows:*

$$
\begin{array}{lcl}
\mathsf{val}(v, l) & = & l(v) \\
\mathsf{val}(c, l) & = & c \\
\mathsf{val}(e_1\ \mathsf{op}\ e_2, l) & = & \mathsf{val}(e_1, l)\ \mathsf{op}\ \mathsf{val}(e_2, l)
\end{array}
$$

Furthermore, notice how Java's $-?-:-$ operator is used in the if-part to merge the state lists generated for both branches. The remainder of Definition 2 should be self-explanatory.

A typical example of a state list in the post state is $\{(a \mapsto a), (b \mapsto a), (c \mapsto d?a : b)\}$ which is desugared in the JML ensures clause `ensures a == \old(a) && b == \old(b) && c = \old(d?a : b)`. Of course, to actually determine if a method is confidential or leaks secret information we also need a secure information policy that labels each variable with a security level.

Next we add `return` statements (without return values) which give rise to an abrupt termination. This means in particular that a method can have multiple exit points. To model this behavior correctly we modify the list data type to a list of lists (denoted with capital letter L) and use the symbol $*$ to represent abrupt termination[2]. Definition 4 gives the specification generation rules when a return statement is added. We have omitted the rules for skip and assignment that stay the same as in Definition 2.

Definition 4 (Rule for non-looping statements, including return)**.**

$$
\begin{array}{l}
\mathsf{sp}(L, \texttt{return}) = [* : L] \\
\mathsf{sp}(L, s_1; s_2) = \\
\quad \texttt{let}\ \ [l_1 : L_1] = \mathsf{sp}(L, s_1) \\
\quad \texttt{in}\ \ \left\{ \begin{array}{ll} \mathsf{sp}([l_1 : L_1], s_2) & \mathit{if}\ l_1 \neq * \\ \mathsf{sp}(L, s_1) & \mathit{otherwise} \end{array} \right. \\
\mathsf{sp}([l : L], \texttt{if}\ (e)\ s_1\ \texttt{else}\ s_2) =
\end{array}
$$

[2]I.e., our list data type is a lifted type with $*$ as bottom type

$$
\begin{cases}
\left[[x_i \mapsto \left\{\begin{array}{ll} l_1[x_i] & \textit{if } l_1[x_i] = l_2[x_i] \\ e \ ? \ l_1[x_i] \ : \ l_2[x_i] & \textit{otherwise} \end{array}\right. \middle| \, i \in \mathbb{N}] : L\right] \\
\qquad \textit{where} \quad \begin{array}{ll} [l_1 : L] = \mathsf{sp}([l : L], s_1) & \textit{with } l_1 \neq * \\ [l_2 : L] = \mathsf{sp}([l : L], s_2) & \textit{with } l_2 \neq * \end{array} \\
\left[[x_i \mapsto \left\{\begin{array}{ll} l_1[x_i] & \textit{if } l_1[x_i] = l_2[x_i] \\ e \ ? \ l_1[x_i] \ : \ l_2[x_i] & \textit{otherwise} \end{array}\right. \middle| \, i \in \mathbb{N}] : [l_1 : L]\right] \\
\qquad \textit{where} \quad \begin{array}{ll} [* : [l_1 : L]] = \mathsf{sp}([l : L], s_1) & \textit{with } l_1 \neq * \\ [l_2 : L] = \mathsf{sp}([l : L], s_2) & \textit{with } l_2 \neq * \end{array} \\
\left[[x_i \mapsto \left\{\begin{array}{ll} l_1[x_i] & \textit{if } l_1[x_i] = l_2[x_i] \\ e \ ? \ l_1[x_i] \ : \ l_2[x_i] & \textit{otherwise} \end{array}\right. \middle| \, i \in \mathbb{N}] : [l_2 : L]\right] \\
\qquad \textit{where} \quad \begin{array}{ll} [l_1 : L] = \mathsf{sp}([l : L], s_1) & \textit{with } l_1 \neq * \\ [* : [l_2 : L]] = \mathsf{sp}([l : L], s_2) & \textit{with } l_2 \neq * \end{array} \\
\left[* : [x_i \mapsto \left\{\begin{array}{ll} l_1[x_i] & \textit{if } l_1[x_i] = l_2[x_i] \\ e \ ? \ l_1[x_i] \ : \ l_2[x_i] & \textit{otherwise} \end{array}\right. \middle| \, i \in \mathbb{N} : L]\right] \\
\qquad \textit{where} \quad \begin{array}{ll} [* : [l_1 : L]] = \mathsf{sp}([l : L], s_1) & \textit{with } l_1 \neq * \\ [* : [l_2 : L]] = \mathsf{sp}([l : L], s_2) & \textit{with } l_2 \neq * \end{array}
\end{cases}
$$

The if-rule becomes so complicated because of the multiple termination modes which leads to four different parts depending on the termination behavior of a branch. The main complication is then how to appropriately merge both branches.

We furthermore add method calls, which we assume to be non-recursive. Definition 5 gives the rule for method calls.

Definition 5 (Non-looping strongest postcondition rules (part 3))**.**

$$
\begin{array}{l}
\mathsf{sp}([l : L], m(\overrightarrow{e})) \ = \\
\quad \text{let} \quad (\overrightarrow{a}, l) = \Delta(m) \\
\qquad\quad l_1 = l\langle \overrightarrow{a} / \overrightarrow{e} \rangle \\
\quad \text{in} \\
\qquad \left[[x_i \mapsto \left\{\begin{array}{ll} l[x_i] & \textit{if } \ l_1[x_i] = x_i \\ l_1[x_i] & \textit{otherwise} \end{array}\right. \middle| \, i \in \mathbb{N}] : L\right]
\end{array}
$$

For method calls the (previously) generated list of the called method is looked up in the context Δ. Note that this also makes the analysis modular.

Together, the functions defined in Definition 2, Definition 4 and Definition 5 can be used to generate a specification pattern for confidentiality in JML.

4.2 Approximate strongest postcondition calculus

The techniques discussed in this section are sound and the specifications can be produced automatically from the Java source code, but in general the results will no longer be precise. That is, in some cases the generated specification pattern will suggest that there is a dependency between variables of security levels Low and High that is in fact not there. The generation of specification patterns for programs that contain while statements thus forms an over-approximation.

The basic idea here is to use JML model fields to express dependencies between fields. The specification pattern generating algorithm will introduce a model field together with its `depends` clause. The user can also chose to manually specify a `represents` clause which has to be verified independently with one of the JML tools. This refinement is optional, the user can chose to not give a represents clause at the cost of more false positives.

In order to deal with model fields, the type of abstract state lists (which used to be $\mathsf{Var} \to \mathsf{Expr}$) is extended to $\mathsf{Var} \to \mathsf{Expr} + \mathcal{P}(\mathsf{Var})$. In this way we can specify the exact expression for normal variables, as well as a finite set of depending variables for model variables. We overload the free variables function FV to take arguments of type $\mathsf{Expr} + \mathcal{P}(\mathsf{Var})$: for expression arguments the function is defined as usual, and for variable set arguments it is simply identity.

The strongest postcondition rule for while statements uses an auxiliary function amv (for "add model variables") which is defined in Definition 6. Its purpose is to add fresh model fields to an abstract state list depending on the variables in the condition and the body of the while. Each potentially assigned variable is paired with its respective model field, which in turn has in its depends clause all possibly depended variables. This is illustrated in Example 3 in Section 6.

Definition 6 (amv function)**.** *The* amv *function (for "Add Model Variables") is inductively defined as:*

$$
\begin{array}{lcl}
\mathsf{amv}(V, l, x := e) & = & l[x \mapsto \_x] :: [\_x \leftarrow \mathrm{FV}(e) \cup V] \\
\mathsf{amv}(V, l, s_1; s_2) & = & \mathsf{amv}(V, \mathsf{amv}(V, l, s_1), s_2) \\
\mathsf{amv}(V, l, \texttt{if } (e) \ s_1 \ \texttt{else } s_2) & = & \\
\multicolumn{3}{l}{\quad [x_i \mapsto \begin{cases} l_1[x_i] & \textit{if } l_1[x_i] = l_2[x_i] \\ l_1[\mathrm{FV}(l_1[x_i])] \cup l_2[\mathrm{FV}(l_2[x_i])] & \textit{otherwise} \end{cases} \Big| \, i \in \mathbb{N}]} \\
\multicolumn{3}{l}{\quad \textit{where} \quad l_1 = \mathsf{amv}(V \cup \mathrm{FV}(e), l, s_1)} \\
\multicolumn{3}{l}{\qquad\qquad\ \ l_2 = \mathsf{amv}(V \cup \mathrm{FV}(e), l, s_2)} \\
\mathsf{amv}(V, l, \texttt{while } (e) \ s) & = & \mathsf{amv}(V \cup \mathrm{FV}(e), l, s)
\end{array}
$$

Note that in case of conditional statements, amv uses double indexing in order to compute the set of variables on which a model variable depends in the pre-state. The notation $l[V]$ is shorthand for $\bigcup\{\mathrm{FV}(l[x]) | x \in V\}$.

The actual rule for generating an appropriate specification pattern for confidentiality is then straightforward. Definition 7 displays the strongest postcondition rules for while statements.

Definition 7 (Approximate strongest postcondition calculus)**.**

$$\mathsf{sp}([l : L], \texttt{while } (e) \ s) \ = \ \mathsf{sp}([\mathsf{amv}(\mathrm{FV}(e), l, s) : L], s)$$

Note that there are still two possible results here, depending on termination behavior of the body of the whole while will terminate normally or with a return statement. Here it also becomes clear that our formalization forms an over-approximation, since the while statement might actually terminate normally or not terminate at all. The remainder of the analysis uses the sp function from the previous section. Model fields are treated exactly the same as the other variables.

In order to analyze the complete Core Java language from Definition 1 we add side-effects in expressions. This is relatively straightforward, because we can syntacticly desugar expressions with side-effects into a statement composed with side-effect free expressions. Adding return values to methods is also straightforward since these are simple bound to the special JML variable `\result`. Because of size constraints we do not elaborate on this further.

4.3 Using existing JML specifications in the specification generation

Existing JML specifications can be used to help guide the specification generation process. We identified a couple of places where this is useful. We do not claim that the list we give is exhaustive, other JML specification constructs can possibly also be used in generating more precise specification patterns.

- **Frame conditions** which are specified in the assignable clause are very useful during program verification, because they limit the parts of the heap that can be modified by a method. In the specification generation all fields that are *not* part of the assignable clause necessarily cannot be interfering. Thus these fields can be ignored when analyzing a method. Similarly we know that with *pure* methods (methods without side effects) one only has to check for non-interference of the return value.
- **Readable if clauses** can be used in JML to specify that a condition (in Java) must be true before the field named in the readable if clause can be read [23, §8.7]. Such clauses can be used to simplify the treatment of branching and refine the treatment of loops. Extending JML with a general readable clause, similar to the assignable clause, can also simplify matters considerable. It seems like a good idea to do this, because readable clauses are helpful with program verification in general [4].
- **Assertions** can be used to specify, for example, that an object (say `obj`) is not a null reference by `//@ assert obj != null`. Such assertions are especially useful when the coverage of non-interference is extended to a termination sensitive variant. In a regular Java program every object can be a null reference and thus possibly throw an exception. This forms in essence another form of branching and it has a similar solution as the if-then-else rule proposed above, including all the bookkeeping *for every object*. If we know for sure that an object is non-null, things are simplified considerably since we only have to consider one execution branch.

  Of course, general assertions can also help, e.g., assertions can state precise information on the relation of a variable with others fields in the pre-state. Which in turns either limits the analysis or can serve as a check and possible refinement at the point of the assertion.

The possibility of using existing specifications to solve multiple problems is always a nice thing. We expect that in practice if formal specification and verification of a non-interference property is necessary then other security and safety properties will also be considered, making (re)use of JML specifications more likely.

5 Soundness of the specification generation

In this section we formalize the notion of non-interference and state that the specification generation algorithm is sound, that is that if the generated specification pattern for confidentiality indicates that a method does not leak information then this is indeed the case.

We assume a denotational semantics $[\![-]\!]$, which maps elements of $\mathsf{Statement}$ to partial functions from State to State. Likewise, we assume a semantics $[\![-]\!]$ exists, which maps elements of Expr to appropriate values. See, for instance, [26]. Furthermore, we assume a context Γ of method declarations, such that $\Gamma(m)$ yields $(\overrightarrow{a}, s)$ where $\overrightarrow{a}$ is a list of formal parameters, and $s \in \mathsf{Statement}$ is the body of the method with name m.

Before we can state soundness of our approach, we first formalize the notion of non-interference introduced in Section 3.

Definition 8 (non-interference)**.** *Given a secure information flow policy* Sif*.*

1. *Let* $s \in \mathsf{Statement}$*, we define (semantical) non-interference to be*

$$\mathsf{nonint}(s, \mathsf{Sif}) = \\ \forall q \in \mathsf{State} \forall p_{\mathsf{High}} : \mathsf{State} \to \mathsf{State} \cdot \left(p_{\mathsf{High}}([\![s]\!]_{p_{\mathsf{High}}(q)}) = p_{\mathsf{High}}([\![s]\!]_q)\right)$$

where $p_{\mathsf{High}}(q)$ *assigns values to variables in* High *independent of* q*, but leaves the values of variables in* Low *unchanged, i.e.*

- $v \in \mathsf{High} \Rightarrow \forall q, q' \in \mathsf{State} \cdot \big(p_{\mathsf{High}}(q)(v) = p_{\mathsf{High}}(q')(v)\big)$,
- $v \in \mathsf{Low} \Rightarrow \forall q \in \mathsf{State} \cdot \big(p_{\mathsf{High}}(q)(v) = q(v)\big)$.

2. *Let* l *be a abstract state list, we define (syntactical) non-interference to be*

$$\mathsf{nonint}(l, \mathsf{Sif}) = \forall v \in \mathsf{Low} \cdot \mathrm{FV}(l(v)) \cap \mathsf{High} = \emptyset$$

3. *Let* L *be a list of abstract state lists, we abuse the above notation to define*

$$\mathsf{nonint}(L, \mathsf{Sif}) = \forall l \in L \cdot \mathsf{nonint}(l, \mathsf{Sif})$$

In the sequel we drop the Sif argument of the nonint predicates, whenever the secure information flow policy is clear from the context.

We prove two theorems. The most important of these is Theorem 4 which gives the soundness result for the specification generation of specification patterns of confidentiality for Core Java.

For non-looping statements, the algorithm generates correct post-conditions. This is formalized in the following lemma.

Lemma 1. *Let* $s \in \mathsf{Statement}$*,* $s \neq \texttt{while}\ (e)\ s$*. There exists an* $l \in \mathsf{sp}(L, s)$ *such that for all* $q \in \mathsf{State}$*,* $v \in \mathsf{Var}$*:*

$$[\![s]\!]_q(v) = [\![l(v)]\!]_q$$

Proof. Induction on the structure of s. □

(Note that since we have not given formal definitions of $[\![-]\!]$ (for brevity's sake), we can only hint at the proof of Lemma 1.)

Theorem 2 (Soundness for non-looping statements)**.** *The strongest postcondition calculus defined by Definition 2, Definition 4 and Definition 5 is sound:*
Let $s \in \mathsf{Statement}$*,* $s \neq \texttt{while}\ (e)\ s$*, then*

$$\mathsf{nonint}(\mathsf{sp}(L, s)) \Rightarrow \mathsf{nonint}(s)$$

Proof.

Suppose $\mathsf{nonint}(\mathsf{sp}(L, s))$
Let $v \in \mathsf{Var}$, $q \in \mathsf{State}$ and $P_{\mathsf{High}} : \mathsf{State} \to \mathsf{State}$.
By Lemma 1 we have an $l \in \mathsf{sp}(L, s)$ with
$[\![s]\!]_q(v) = [\![l(v)]\!]_q$ and
$[\![s]\!]_{p_{\mathsf{High}}(q)}(v) = [\![l(v)]\!]_{p_{\mathsf{High}}(q)}$.
If $v \in \mathsf{Low}$ then
since no variables from High occur in $\mathrm{FV}(l(v))$, $[\![l(v)]\!]_q = [\![l(v)]\!]_{p_{\mathsf{High}}(q)}$.
Therefore $p_{\mathsf{High}}([\![s]\!]_q(v)) = p_{\mathsf{High}}([\![s]\!]_{p_{\mathsf{High}}(q)})(v)$.
If $v \in \mathsf{High}$ then
the values of $p_{\mathsf{High}}([\![s]\!]_q)(v)$ and $p_{\mathsf{High}}([\![s]\!]_{p_{\mathsf{High}}(q)})(v)$ are completely determined by p_{High} (independent of $[\![s]\!]_q$ and $[\![s]\!]_{p_{\mathsf{High}}(q)}$) and are therefore equal.
Thus, $p_{\mathsf{High}}([\![s]\!]_{p_{\mathsf{High}}(q)}) = p_{\mathsf{High}}([\![s]\!]_q)$.

□

In order to prove soundness for all strongest postcondition rules lemma 1 needs to be weakened:

Lemma 3. *Let* $s \in \mathsf{Statement}$. *There exists an* $l \in \mathsf{sp}(L, s)$ *such that for all* $q \in \mathsf{State}$, $v \in \mathsf{Var}$*: If* s *terminates (when started from* q*), then*

1. *if* $l(v) \in \mathsf{Expr}$, *then* $[\![s]\!]_q(v) = [\![l(v)]\!]_q$,
2. *if* $l(v) \subseteq \mathsf{Var}$, *then* $\exists e \in \mathsf{Expr} \cdot \big(\mathrm{FV}(e) \subseteq l(v) \wedge [\![s]\!]_q(v) = [\![e]\!]_q\big)$.

Proof. Induction on the structure of s. The case for while uses induction to the number of times the body is executed. □

(Note that since we have not given formal definitions of $[\![-]\!]$ (for brevity's sake), we can only hint at the proof of Lemma 3.)

The actual soundness prove for all the strongest postcondition rules is then straightforward:

Theorem 4 (Soundness of all rules)**.** *The strongest postcondition rules defined by Definition 2, Definition 4, Definition 5 and Definition 7 are sound:*
Let $s \in \mathsf{Statement}$, *then*

$$\mathsf{nonint}(\mathsf{sp}(L, s)) \Rightarrow \mathsf{nonint}(s)$$

Proof. Analogous to Theorem 2. □

6 Specification generation examples

In this section the JML specification generation is illustrated with two examples: an example with and one without a loop.

6.1 Non-looping specification generation

Example 2 contains two methods: `int decrementhigh(int i)` and `void m()` (the latter calls `decrementhigh`). There are only two fields, `high` and `low` which have security level High and Low respectively.

For clarity we show the full `ensures` clause –including the expressions for `high`, which are not necessary for expressing confidentiality– and the state lists in the pre-and postcondition for both methods.

Example 2.

```
  int high,low;  // high:H, low:L

// pre  : [{(low,low), (high,high), (i,i)}]

//@ assignable high;
//@ ensures \result == \old(i);
//@ ensures high == \old(high - 1);
  int decrementhigh(int i){
      high = high -1;
      return i;}
// post : [*, {(low,low), (high,high - 1), (i,i), (\result,i)}]

// pre  : [{(low,low), (high,high)}]

//@ assignable low,high;
//@ ensures low == \old(high);
```

```
//@ ensures high == \old(high - 1);
void m(){
      low = decrementhigh(high);}
// post : [{(low,high), (high,high - 1)}]
```

The specification of method `decrementhigh(int i)` follows directly from the rules. Method `m()` is more interesting, because the generated list of the other method is used in the method call. Notice that since `high` is passed along as a parameter to method `decrementhigh` there is a dependency between field `low` and the value of `high` in the pre-state and the method thus leaks information.

6.2 Approximate specification generation

Example 3 illustrates JML specification generation –using the specification pattern for confidentiality– for a method with a loop.

Example 3.

```
//@ model int _high, _low;
//@ depends _high <- high
//@ depends _low <- high,low;
//@ represents _low <- high;    // optional refinement by user

int high,low; // high:H low:L

//@ requires high > 0;
//@ assignable high, low;
//@ ensures low == \old(_low);  // low == \old(high);
    void m() {
      low=0;
      while (high > 0){
        high--;
        low++; } }
```

We assume that the `requires` clause is given, the `assignable` clause is added for completeness sake, it does not play a role in the actual specification generation. Note that the method leaks information; after evaluation the field `low` will have the value of field `high` in the pre-state which represents an illegal flow of information from security level High to level Low.

The JML specification generation for `while` statements will automatically generate the `model`, `depends` and `ensures` clauses. The JML specification is still partial since the model fields[3] do not have an explicit `represents` clause. The specification does suggest that there is a problem since field `low` is equal to the value of the model field `_low` in the pre-state and `_low` depends on variable `high` (and `low`), thus representing an illegal flow of information that breaks confidentiality.

However, our analysis method is an over-approximation, so in principle it is possible that method `m()` does not leak any information. Indeed, for this reason the user can add a `represents` clause to verifying that `_low` depends on the value of field `high` in the pre-state, as is indicated in Java comment in the example.

[3]We use the convention that model fields always start with an underscore '_'. This makes it easier to distinguish them from normal fields and also makes implicit relations between model and normal field more transparent, i.e., `x` and `_x` are related.

7 Towards termination sensitive non-interference

This paper focuses on *termination-insensitive* non-interference, meaning that the non-interference property (confidentiality, integrity) is only guaranteed if the analyzed program terminates normally. If the program terminates with an exception or does not terminate at all (hangs), non-interference is no longer assured.

The specification pattern for confidentiality can easily be used in `signals` clauses as well. When used in this way, it expresses that if an exception of a certain type is thrown then the non-interference property holds. The specification generation algorithm can be extended to incorporate exceptions and use such `signals` clauses at the cost of more complex rules, especially involving more bookkeeping, since the termination behavior needs to be encoded into the abstract state. Exceptions in Java also complicate the control flow mechanism, since they can be caught and thus do not have to show at the outside of a method.

Similarly, it is possible to use the specification pattern for confidentiality in JML's `diverges` clause. Such a specification pattern states for which *precondition* a non-interference result holds provided the method does not terminate. Such diverges clauses have to be given by the user since determining if a program terminates is undecidable in general.

Termination behavior itself can also leak information, as illustrated by Example 4:

Example 4.

```
boolean high; // high:H

public void m(){
     if(high) throw new Exception();
}
```

Assuming the termination behavior of a program is observable, the value of field `high` is leaked. Such cases of information leakage are not covered by the specification pattern for confidentiality. Termination sensitive non-interference can easily be specified in JML using the `normal_behavior` and `exceptional_behavior` keywords, which specify that the method *must* terminate normally or with an exception respectively. In effect this makes termination sensitive and insensitive termination coincide, because only one termination mode per method is allowed.

More generally, termination sensitive non-interference in JML can be specified using a JML `ghost` field, as illustrated by Example 5:

Example 5.

```
boolean high;  // high: H;

//@ public ghost int _tmode;

/*@ invariant _tmode == 0 || // normal termination
              _tmode == 1 || // exceptional termination
              _tmode == 2;   // non-termination
 */

//@ ensures  \old(high) ==> _tmode == 1;
```

```
//@ ensures !\old(high) ==> _tmode == 0;
public void m() throws Exception{
    if (high){
        //@ set _tmode = 1;
        throw new Exception();
    }
    //@ set _tmode = 0;
}
```

By encoding the termination behavior of a method in the ghost field we can specify that the value of field `high` in the pre-state determines the termination behavior of the method. Thus information is leaked via the termination behavior of method `m`(). Notice how the invariant is used to guarantee that there are only three termination modes.

The specification pattern for confidentiality is no longer used. A specification that has the ambition to specify termination sensitive non-interference in JML should use the specification pattern for confidentiality and should specify how the value of each High field in pre-state influences termination behavior of a method. Automatically generating termination-sensitive non-interference specifications is left for future work.

8 Related work

In the context of the *SecSafe* project [13] several security properties which are relevant for Java Card applets have been identified [24], they concern amongst others the absence of certain exception types at the top-level, atomicity of updates, no unwanted overflow, only memory allocation during the install phase of the applet and conditional execution points. All these points can either be expressed directly in JML or in the underlying semantic model used by one of the JML tools, as is shown by several researchers [25, 27, 17, 18, 16]. Expressing non-interference properties like confidentiality and integrity directly in JML has, to the best of our knowledge, not been done before.

The work of Dufay, Felty and Matwin [10] is probably most related to ours. They add keywords to JML that express confidentiality and have modified the Krakatoa tool [8] in order to prove non-interference properties for this extended version of JML. The main disadvantages of this approach is that only the (modified) Krakatoa tool can be used to prove their extended JML annotations. Thus making one of JML's most attractive features –its tool-support– obsolete.

The group behind the KeY-tool [2] uses a similar approach to ours (also based on Joshi and Leino's paper [20]) to prove confidentiality [9]. The main difference is that they do not use a separate specification language but express confidentiality directly in the dynamic logic which they use to reason (interactively) about sequential Java (Card) programs.

Secure information flow is still a very active research field, most other related work is based on security typing [29]. An overview of the field can be found in [28].

The notion of a specification pattern originated in the work of Dwyer *et. al.* [11]. They noticed that patterns emerge when specifying temporal properties for concurrent systems. To the best of our knowledge specification patterns for JML have not been proposed before.

Other tools that automatically generate JML specifications are Daikon [12] and Houdini [14]. Both tools take no input (besides the source of the to be specified program) and use heuristics to generate likely invariants and pre- and postconditions

for programs. Such tools are usually used to generate a basic specification skeleton that has to be improved manually by the user.

9 Conclusions and future work

We show the feasibility of applying JML for specifying non-interference properties like confidentiality and integrity using specification patterns for JML. A sound algorithm for automatic generation of such specifications is introduced. Existing JML specifications can be used to generate more precise specifications. Moreover, it is possible and straightforward for users to refine the generated specifications.

For future work we intend to implement the specification generation algorithm and extend it to support all parts of sequential Java. We also want to investigate if more complex secure information flow policies, e.g., with declassification mechanisms, can be used as a basis for generating JML specifications. Finally, generating JML specifications that express stronger forms of non-interference, e.g., taking into account termination behavior, is another research challenge.

References

[1] J. Agat. Transforming out Timing Leaks. In *27th ACM Symposium on Principles of Programming Languages*, pages 40–53. ACM Press, Jan. 2000.

[2] W. Ahrendt, T. Baar, B. Beckert, R. Bubel, M. Giese, R. Hähnle, W. Menzel, W. Mostowski, A. Roth, S. Schlager, and P. H. Schmitt. The KeY tool. *Software and System Modeling*, 2004. Online First issue, to appear in print.

[3] G. Barthe, T. Rezk, and M. Warnier. Preventing timing leaks through transactional branching instructions. In *3th Workshop on Quantitative Aspects of Programming Languages Proceedings,to appear*. Elsevier Science, 2005.

[4] J. Boyland. Why we should not add `readonly` to Java (yet). In *Informal proceedings of the 7th Workshop on Formal Techniques for Java-like Programs (FTfJP)*, 2005.

[5] C.-B. Breunesse and E. Poll. Verifying JML specifications with model fields. In *Formal Techniques for Java-like Programs. Proceedings of the ECOOP'2003 Workshop*, 2003.

[6] L. Burdy, Y. Cheon, D. Cok, M. Ernst, J. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll. An overview of JML tools and applications. *International Journal on Software Tools for Technology Transfer (STTT)*, 2004. To appear.

[7] Y. Cheon, G. Leavens, M. Sitaraman, and S. Edwards. Model variables: cleanly supporting abstraction in design by contract. *spe*, 35(6):583–599, may 2005.

[8] E. Contejean, J. Duprat, J.-C. Filliâtre, C. Marché, C. Paulin-Mohring, and X. Urbain. The Krakatoa tool for certification of Java/Java Card programs annotated in JML. *Journal of Logic and Algebraic Programming*, 58(1-2):89–106, 2004. Available via the Krakatoa home page at http://www.lri.fr/~marche/krakatoa/.

[9] Á. Darvas, R. Hähnle, and D. Sands. A Theorem Proving Approach to Analysis of Secure Information Flow. In *Proc. 2nd International Conference on Security in Pervasive Computing*, LNCS. Springer-Verlag, to appear, 2005.

[10] G. Dufay, A. Felty, and S. Matwin. Privacy-Sensitive Information Flow with JML. In *Conference on Automated Deduction (CADE) proceeedings*, LNCS. Springer-Verlag, 2005.

[11] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Patterns in Property Specification for Finite-State Verification. In *Proceedings of the 21st International Conference on Software Engineering (ICSE '99)*. ACM, may 1999.

[12] M. Ernst, J. Cockrell, W. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEETSE*, 27(2):99–123, 2001.

[13] European IST-1999-29075 Project SecSafe. http://www.doc.ic.ac.uk/~siveroni/secsafe/.

[14] C. Flanagan, K. Leino, M. Lillibridge, G. Nelson, J. 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.

[15] J. Goguen and J. Meseguer. Security policies and security models. In *IEEE Symp. on Security and Privacy*, pages 11–20. IEEE Comp. Soc. Press, 1982.

[16] E. Hubbers, M. Oostdijk, and E. Poll. Implementing a formally verifiable security protocol in Java Card. In D. Hutter, G. Müller, W. Stephan, and M. Ullmann, editors, *Proceedings of the 1st International Conference on Security in Pervasive Computing*, volume 2802 of *LNCS*, pages 213–226. Springer-Verlag, 2004.

[17] B. Jacobs, C. Marche, and N. Rauch. Formal verification of a commercial smart card applet with multiple tools. In C. Rattray, S. Maharaj, and C. Shankland, editors, *Algebraic Methodology and Software Technology (AMAST'04)*, volume 3116 of *LNCS*, pages 21–22. Springer, Berlin, 2004.

[18] B. Jacobs, M. Oostdijk, and M. Warnier. Source Code Verification of a Secure Payment Applet. *Journ. of Logic and Algebraic Programming*, 58(1-2):107–120, 2004.

[19] JML web site. http://www.jmlspecs.org.

[20] R. Joshi and K. Leino. A semantic approach to secure information flow. *Science of Comput. Progr.*, 37(1-3):113–138, 2000.

[21] B. W. Lampson. A note on the Confinement Problem. *Communications of the ACM*, 16(10):613–615, 1973.

[22] G. T. Leavens, A. L. Baker, and C. Ruby. JML: A Notation for Detailed Design. In H. Kilov, B. Rumpe, and W. Harvey, editors, *Behavioral Specification for Businesses and Systems*, chapter 12, pages 175–188. Kluwer Academic Publishers, 1999.

[23] G. T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. Cok, and J. Kiniry. *JML reference Manual (draft)*, 2005. available at: http://www.jmlspecs.org/jmlrefman/jmlrefman_toc.html.

[24] R. Marlet and D. L. Métayer. Security properties and Java Card specificities to be studied in the SecSafe project. Technical Report SECSAFE-TL-006, Trusted Logic, Aug. 2001.

[25] W. Mostowski. Formalisation and verification of Java Card security properties in Dynamic Logic. In M. Cerioli, editor, *Proceedings, Fundamental Approaches to Software Engineering (FASE) Conference 2005, Edinburgh, Scotland*, volume 3442 of *LNCS*, pages 357–371. Springer, April 2005.

[26] H. R. Nielson and F. Nielson. *Semantics with Applications*. Wiley Professional Computing, 1992.

[27] M. Pavlova, G. Barthe, L. Burdy, M. Huisman, and J.-L. Lanet. Enforcing High Level Security Properties For Applets. In J.-J. Quisquater, P. Paradinas, Y. Deswarte, and A. El Kalam, editors, *Proceedings of Smart Card Research and Advanced Applications (CARDIS)*. IFIP, Kluwer Academic Publishers, 2004.

[28] A. Sabelfeld and A. C. Myers. Language-Based Information-Flow Security. *IEEE Journal on selected areas in communications*, 21(1), 2003.

[29] D. Volpano and G. Smith. A Type-Based Approach to Program Security. In *Proc. 7th Int'l Joint Conference on the Theory and Practice of Software Development*, volume 1214 of *LNCS*, pages 607–621. Springer, 1997.