A Formal Connection between Security Automata and JML Annotations*

Marieke Huisman[1] and Alejandro Tamalet[2]

[1] University of Twente, Netherlands
[2] University of Nijmegen, Netherlands

Abstract. Security automata are a convenient way to describe security policies. Their typical use is to monitor the execution of an application, and to interrupt it as soon as the security policy is violated. However, run-time adherence checking is not always convenient. Instead, we aim at developing a technique to verify adherence to a security policy statically. To do this, we consider a security automaton as specification, and we generate JML annotations that inline the monitor – as a specification – into the application. We describe this translation and prove preservation of program behaviour, *i.e.*, if monitoring does not reveal a security violation, the generated annotations are respected by the program.
The correctness proofs are formalised using the PVS theorem prover. This reveals several subtleties to be considered in the definition of the translation algorithm and in the program requirements.

1 Introduction

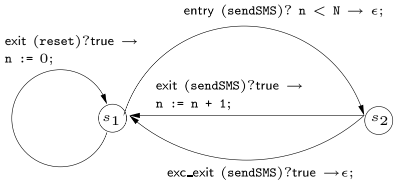
With the emergence of a new generation of trusted personal devices (mobile phones, PDAs, *etc.*), the demand for techniques to guarantee application security has become even more prominent. A common approach is to monitor executions with a security automaton [13]. Upon entry or exit of a security-critical method, the security automaton updates its internal state. If it reaches an "illegal" state, the application will be stopped and a security violation will be reported. This approach is particularly suited for properties that are expressed as sequences of legal method calls, such as life cycle properties, or constraints that express how often or under which conditions a method can be called. However, such a monitoring approach is not suited for all applications, depending on their nature and use; sometimes statical means to enforce security are necessary.

Security experts typically express security requirements by a collection of security automata or temporal logic formulae. However, many program verification tools use a Hoare logic style for the specifications (*i.e.*, pre- and postconditions). Therefore, as a first step towards static verification of such security properties, this paper proposes a translation from security properties expressed as an automaton into program annotations.

* This work is partially funded by the IST FET programme of the European Commission, under the IST-2005-015905 Mobius project. Research done while the authors where at INRIA Sophia Antipolis.

The translation in this paper is defined for Java programs. It is defined in several steps. For each step we provide a correctness proof. (*i*) We translate a *partial automaton* to a *total automaton* that contains a special trap state to model that an error has occurred. We show that the behaviour of a program monitored with a partial automaton is equivalent to the behaviour of the program monitored with the total automaton. (*ii*) Using an extension of JML [9], we generate annotations that capture the behaviour of the total automaton. These are special method-level set-annotations that are evaluated upon entry or exit of a method. We show that run-time monitoring of the program only throws a (new) exception to signal an annotation violation if the monitor reaches the trap state, otherwise the annotated program has the same executions as the monitored program. (*iii*) We inline the set-annotations from the method specification to the method body and prove equivalence of the run-time checking behaviour. All results in the paper have been established formally using the PVS theorem prover [11]. The complete formalisation is available via `http://www.cs.ru.nl/~tamalet/`.

To prove correctness, the order in which method specifications are evaluated is important. Further, we had to add an explicit requirement that `finally` blocks could not override annotation violation exceptions thrown inside `try` or `catch` statements (see also [8]). The last complication that we encountered was how to specify conveniently that specification-only constructs and steps taken by the monitor did not have any side-effects on the program state. More detailed information about the proofs is given in Section 4.



Automaton vars = $\{n\}$ Program vars = $\emptyset$

Fig. 1. Example Property Automaton

Throughout this paper, we use the *limited SMS* example property of Fig. 1 (where ϵ denotes a skip) to illustrate the different translations: the method `sendSMS` can be called and terminate successfully at most `N` times in between calls to `reset`. The counter is not increased if `sendSMS` terminates because of an uncaught exception (with label exc_exit(sendSMS)), and `reset` should not be called from within `sendSMS`. Even though very basic, this example is representative of a wide range of important resource-related security properties.

The rest of this paper is organised as follows. Section 2 formalises the automaton format and defines completion. Next, Section 3 defines the semantics of monitored and annotated programs. Section 4 defines the translation and proves correctness. Sections 5 and 6 discuss related and future work and conclusions.

2 Modelling Security Properties with Automata

The automata that we use to express security properties are called Property Automata (PA). These are extended finite state machines particularly suited for monitoring, since transitions do not only depend on the automaton's state (*i.e.*, the current control point and a valuation for the automaton's variables), but also on the state of the monitored program. Transitions are labelled with guards, events and a list of actions. Events specify the method whose entry and/or exit is being monitored, with a distinction between normal and exceptional exits. Guards describe the conditions under which a transition can be applied. They depend on (*i*) the automaton state, (*ii*) the state of the program that is being monitored, and (*iii*) the argument of the method, in case the event is method entry; the result of the method, in case the event is normal method exit; or the exception with which the method returns, in case the event is exceptional method exit. Actions describe how the automaton state is updated by a transition.

Throughout, we assume that CP and $\mathcal{N}$ are possibly infinite, but countable non-empty sets of control points and names. PA and programs share the definitions of values, types and exceptions, denoted $\mathcal{V}$, $\mathcal{T}$ and $\mathcal{E}$, respectively. These are defined by the following grammar, where $\mathbb{B}$ and $\mathbb{Z}$ denote the standard sets of booleans and integers, respectively[1].

$$\begin{aligned}
\mathcal{V} &= \mathsf{B}(b : \mathbb{B}) \mid \mathsf{I}(i : \mathbb{Z}) \mid \mathsf{Null} \mid \mathsf{R}(i : \mathbb{Z}) \mid \mathbb{1} \mid \bot \\
\mathcal{T} &= \mathsf{Bool} \mid \mathsf{Int} \mid \mathsf{Ref} \mid \mathsf{Void} \\
\mathcal{E} &= \mathsf{Throwable} \mid \mathsf{RunTimeException} \mid \mathsf{JMLException}
\end{aligned}$$

The type Void, inhabited by $\mathbb{1}$, models methods without results; a reference can be Null or contain a number representing the location where the object is stored; $\bot$ is used to denote the outcome of an expression whose evaluation is undefined (in Java this would typically result in an exception).

A PA consists of (*i*) a name, (*ii*) a class name, to specify which class is being monitored, (*iii*) a finite set of control points, (*iv*) an initial control point, (*v*) a set of events, to specify which methods are being monitored, (*vi*) a set of PA variable declarations, to describe the internal state of the automaton, (*vii*) a set of program variable declarations, to specify which program variables will be inspected by the monitor, and (*viii*) a set of transitions. Transitions relate source and target control points; they are labelled with events, a guard and a list of actions. An event is a tuple of an event type (entry, exit or exceptional exit), and a method name. Each action assigns the result of an expression (containing both program and PA variables) to a PA variable. Notice that we only monitor classes here. This is often the case in practice, because security-critical methods are often static API methods. However, a more precise formalisation of Java's semantics would allow to monitor objects as well. Formally, a PA is defined as follows.

$$\begin{aligned}
\mathit{Decl} &= \texttt{[\#}\ \mathsf{type} : \mathcal{T}, \mathsf{name} : \mathcal{N}, \mathsf{init} : \mathcal{V}\ \texttt{\#]} \\
\mathit{Event} &= \texttt{[\#}\ \mathsf{etype} : (\mathsf{entry} \mid \mathsf{exit} \mid \mathsf{exc\_exit}), \mathsf{mname} : \mathcal{N}\ \texttt{\#]} \\
\mathit{Trans} &= \texttt{[\#}\ \mathsf{source}, \mathsf{dest} : \mathit{CP}, \mathsf{event} : \mathit{Event}, \mathsf{action} : (\texttt{[\#}\,\mathsf{target} : \mathcal{N}, \mathsf{expr} : \mathit{Expr}\ \texttt{\#]})^*, \\
&\qquad\ \ \mathsf{guard} : \mathit{PAState} \times \mathit{PState} \times (\mathcal{V} \mid \mathcal{E}) \to \mathbb{B}\ \texttt{\#]} \\
\mathit{PA} &= \texttt{[\#}\ \mathsf{name}, \mathsf{clname} : \mathcal{N}, \mathsf{cps} : \mathcal{P}(\mathit{CP}), \mathsf{init} : \mathit{CP}, \mathsf{events} : \mathcal{P}(\mathit{Event}), \\
&\qquad\ \ \mathsf{pa\_var\_decl} : \mathcal{P}(\mathit{Decl}), \mathsf{prog\_var\_decl} : \mathcal{P}(\mathit{Decl}), \mathsf{trans} : \mathcal{P}(\mathit{Trans})\ \texttt{\#]}
\end{aligned}$$

[1] We will use a PVS-like notation to declare abstract data types and records (enclosed by [# and #]). Further, if x is a record with field y, we use x.y to access field y, and x(# y := z #) to denote the record x with the field y updated to z.

We require a PA to be *deterministic*, *i.e.*, for every source control point and event there is always at most one guard that holds. A PA is *total* if for any source control point and event, there is always a guard that holds; otherwise it is *partial.* Every deterministic PA can be completed into a total one (by function complete): add a special control point halted, together with transitions for every control point and every event to halted, where the guard is the negation of the disjunction of all other guards for this control point and event. Additionally, add unconditional transitions from halted to halted for every possible event.

A PA is *wellformed* if: (*i*) variable names are unique and are not reserved words, (*ii*) guards do not have side-effects, (*iii*) guards and actions only use declared variables, and (*iv*) control points and events in transitions are declared.

The state of a PA consists of a current control point, and the store of automaton variables (the program store is not part of the automaton state): $\mathit{PAState} = \texttt{[\#}\,\mathsf{current} : \mathit{CP}, \mathsf{store}_{\mathsf{A}} : \mathit{Store}\ \texttt{\#]}$. Given PA a, the transition function $\mathit{\Delta}_a$ specifies how an automaton state σ_A is updated for a given program state σ_P, an event e, and a value or exception v (where ε is the arbitrary choice operator, and apply is a function that updates the automaton store according to a list of actions in the obvious way).

$$\begin{array}{l}
\mathit{\Delta}_a : \mathit{PAState} \times \mathit{PState} \times \mathit{Event} \times (\mathcal{V} \mid \mathcal{E}) \hookrightarrow \mathit{PAState} \\
\mathit{\Delta}_a(\sigma_A, \sigma_P, e, v) = \\
\quad \mathsf{let}\ t = \varepsilon(\{t \in \mathsf{trans}(a) \mid t.\mathsf{source} = \sigma_A.\mathsf{current} \wedge t.\mathsf{event} = e \wedge \\
\qquad\qquad\qquad\qquad\qquad\qquad t.\mathsf{guard}(\sigma_A.\mathsf{store}_{\mathsf{A}}, \sigma_P.\mathsf{fields.store}, v)\})\ \mathsf{in} \\
\qquad \texttt{(\#}\,\mathsf{current} := t.\mathsf{dest}, \mathsf{store}_{\mathsf{A}} := \mathsf{apply}(t.\mathsf{action}, \sigma_A.\mathsf{store}_{\mathsf{A}}, \sigma_P.\mathsf{fields.store})\ \texttt{\#)}
\end{array}$$

In a total PA a, the transition function $\mathit{\Delta}_a$ is total. A partial automaton gets stuck on a certain input if and only if the completed PA reaches the state halted.

$$\mathit{\Delta}_a(\sigma_A, \sigma_P, e, v) = \perp \Leftrightarrow \mathit{\Delta}_{\mathsf{complete}(a)}(\sigma_A, \sigma_P, e, v).\mathsf{current} = \mathsf{halted} \tag{1}$$

Example The property specified in Fig. 1 is encoded by the following PA[2], while Fig. 2 shows the completed PA (where new transitions are dashed).

$$\begin{array}{l}
\texttt{(\#}\ \mathsf{name} := \mathrm{LimitSMS}, \mathsf{clname} := \texttt{Messaging}, \mathsf{cps} := \{s_1, s_2\}, \mathsf{init} := s_1, \\
\quad \mathsf{events} := \{\texttt{(\#}\,\mathsf{etype} := e, \mathsf{mname} := \texttt{sendSMS \#)} \mid e \in \{\mathsf{entry}, \mathsf{exit}, \mathsf{exc\_exit}\}\} \cup \\
\qquad\qquad \{\texttt{(\#}\,\mathsf{etype} := \mathsf{exit}, \mathsf{mname} := \texttt{reset \#)}\}, \\
\quad \mathsf{pa\_var\_decl} := \{\texttt{(\#}\,\mathsf{name} := n, \mathsf{type} := \mathsf{Int}, \mathsf{init} := 0\ \texttt{\#)}\}, \mathsf{prog\_var\_decl} := \emptyset, \\
\quad \mathsf{trans} := \{\ \texttt{(\#}\,\mathsf{source} := s_1, \mathsf{dest} := s_2, \mathsf{guard} := \lambda(\sigma_A, \sigma_P, v).\sigma_A(n) < N, \\
\qquad\qquad\qquad \mathsf{event} := \texttt{(\#}\,\mathsf{etype} := \mathsf{entry}, \mathsf{mname} := \texttt{sendSMS \#) \#)}, \\
\qquad\qquad \texttt{(\#}\,\mathsf{source} := s_2, \mathsf{dest} := s_1, \mathsf{action} := [\texttt{(\#}\,\mathsf{target} := n, \mathsf{expr} := n + 1\ \texttt{\#)}] \\
\qquad\qquad\qquad \mathsf{event} := \texttt{(\#}\,\mathsf{etype} := \mathsf{exit}, \mathsf{mname} := \texttt{sendSMS \#) \#)}, \\
\qquad\qquad \texttt{(\#}\,\mathsf{source} := s_2, \mathsf{dest} := s_1, \\
\qquad\qquad\qquad \mathsf{event} := \texttt{(\#}\,\mathsf{etype} := \mathsf{exc\_exit}, \mathsf{mname} := \texttt{sendSMS \#) \#)}, \\
\qquad\qquad \texttt{(\#}\,\mathsf{source} := s_1, \mathsf{dest} := s_1, \mathsf{action} := [\texttt{(\#}\,\mathsf{target} := n, \mathsf{expr} := 0\ \texttt{\#)}], \\
\qquad\qquad\qquad \mathsf{event} := \texttt{(\#}\,\mathsf{etype} := \mathsf{exit}, \mathsf{mname} := \texttt{reset \#) \#)}\}\ \texttt{\#)}
\end{array}$$

[2] Where we leave the default guard $\lambda(\sigma_A, \sigma_P, v).\mathsf{true}$ and empty action ϵ implicit.

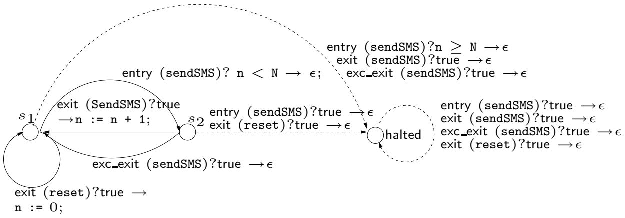


Fig. 2. Automaton of Fig. 1, after completion

$$
\begin{aligned}
\mathit{Expr} = {} & \mathsf{Plus}(n_1, n_2 : \mathit{Expr}) \mid \mathsf{Var}_{\mathbb{I}}(n : \mathcal{N}) \mid \mathsf{Not}(b : \mathit{Expr}) \mid \mathsf{And}(b_1, b_2 : \mathit{Expr}) \mid \\
& \mathsf{Eq}(e_1, e_2 : \mathit{Expr}) \mid \mathsf{Var}_{\mathbb{B}}(n : \mathcal{N}) \mid \mathsf{Var}_{\mathbb{R}}(n : \mathcal{N}) \mid \mathsf{CondExpr}(c, e_1, e_2 : \mathit{Expr}) \mid \\
& \mathsf{Assign}(n : \mathcal{N}, e : \mathit{Expr}) \mid \mathsf{Call}(o : \mathit{Expr}, mn : \mathcal{N}, p : \mathit{Expr}) \mid \mathsf{Const}(v : \mathcal{V}) \\
\mathit{Stmt} = {} & \mathsf{Skip} \mid \mathsf{Sequence}(s_1, s_2 : \mathit{Stmt}) \mid \mathsf{IfThenElse}(c : \mathit{Expr}, s_1, s_2 : \mathit{Stmt}) \mid \\
& \mathsf{While}(c : \mathit{Expr}, s : \mathit{Stmt}) \mid \mathsf{StmtExpr}(e : \mathit{Expr}) \mid \mathsf{Throw}(e : \mathcal{E}) \mid \\
& \mathsf{TryCatchFinally}(t : \mathit{Stmt}, e : \mathcal{E}, c, f : \mathit{Stmt}) \mid \mathsf{Set}(n : \mathcal{N}, e : \mathit{Expr}) \mid \\
& \mathsf{CaseSet}(b : \mathsf{list}[\mathit{Expr} \times \mathit{Stmt}]) \mid \mathsf{Assert}(e : \mathit{Expr})
\end{aligned}
$$

Fig. 3. Abstract syntax of expressions and statements

3 Programs and Semantics

This section first defines an abstract syntax of programs, followed by their semantics. Both are fairly standard, except that the semantics is parametrised on the treatment of specifications. In particular, we define a run-time checking and a monitoring semantics, that evaluate differently upon method call and exit.

3.1 Program Syntax

Our language is a restricted subset of (sequential) Java, abstracting away from typical object-oriented features, and in particular from method resolution; instead we assume that the annotated class contains method bodies for the relevant methods, thus method lookup is trivial. We consider only a few exceptions, and assume that methods have only one parameter. We believe, however, that our formalisation contains all constructs that are relevant for proving correctness of our inlining algorithm for class-based monitoring, and implementing the algorithm for the full language is mainly an engineering issue.

Figure 3 defines expressions and statements (we use the term *body* to denote either an expression or a statement), *e.g.*, Call represents a call to method mn on target o with argument p. Notice that we define several special language constructs to represent JML annotations: Set, to update ghost variables (*i.e.*, specification-only variables), CaseSet, to abbreviate a list of conditional ghost variable updates, and Assert, to evaluate a condition on the program state. A standard program semantics ignores these statements, whereas the annotated program semantics evaluates them.

$$\begin{aligned}
\mathit{Method} &= [\#\ \mathsf{name} : \mathcal{N}, \mathsf{param} : \mathit{Decl}, \mathsf{lvars} : \mathcal{P}(\mathit{Decl}), \mathsf{body} : \mathit{Stmt},\\
&\qquad \mathsf{res} : \mathit{Expr}, \mathsf{res\_type} : \mathcal{T}, \mathsf{pre}, \mathsf{post} : \mathit{Expr} \rightarrow \mathit{Expr},\\
&\qquad \mathsf{pre\_set}, \mathsf{post\_set} : \mathit{Expr} \rightarrow \mathit{Stmt}, \mathsf{exc\_set} : \mathcal{E} \rightarrow \mathit{Stmt}\ \#]\\
\mathit{Class} &= [\#\ \mathsf{name} : \mathcal{N}, \mathsf{super} : \mathcal{N}_{\perp}, \mathsf{fields} : \mathcal{P}(\mathit{Decl}), \mathsf{methods} : \mathcal{P}(\mathit{Method}),\\
&\qquad \mathsf{inv} : \mathit{Expr}, \mathsf{ghost\_vars} : \mathcal{P}(\mathit{Decl})\ \#]\\
\mathit{Program} &= [\#\ \mathsf{classes} : \mathcal{P}(\mathit{Class})\ \#]
\end{aligned}$$

Fig. 4. Abstract Syntax for Programs

Figure 4 describes the syntax for methods, classes and programs. To ensure that every method has an appropriate return expression, it is part of the method signature. Furthermore, methods can be annotated with pre- and postconditions, and classes with invariants. To support our annotation generation algorithm, we define special annotations called pre_set, post_set and exc_set. These annotations describe the updates to the ghost variables at method entry, exit and exceptional exit, respectively. Pre- and postcondition and the different method specification-level set annotations have a function type to allow the use of the method parameter, the method result, or the returned exception, respectively.

A program is said to be *wellformed* if (*i*) names of fields, local variables and ghost variables are disjoint and are not reserved words; (*ii*) class names are unique; (*iii*) method names are unique; (*iv*) every variable name that is used is declared; and (*v*) only ghost variables are the target of Set statements.

3.2 Natural Semantics

The behaviour of a program is described via a big step semantics. We closely follow Von Oheimb's formalisation of Java [10], with simplifications wherever possible, due to our simplified program syntax. A judgement $P \vdash \langle e, \sigma \rangle \triangleright \langle v, \sigma' \rangle$ means that the body e evaluates to v, while transforming the state σ into σ', in the context of the program P. Note that v is $\mathbb{1}$ for normally terminating *statements*, while v is $\perp$ whenever evaluation finishes in an exceptional state.

A basic program state *PState* is composed of an optional exception and a store. The store maps every field and local variable to a value.

$$\begin{aligned}
\mathit{PState} &= [\#\ \mathsf{exc} : \mathit{Excp}_{\perp}, \mathsf{store} : \mathit{PStore}\ \#]\\
\mathit{PStore} &= [\#\ \mathsf{fields} : \mathcal{N} \mapsto \mathcal{V}, \mathsf{loc\_vars} : \mathcal{N} \mapsto \mathcal{V}\ \#]
\end{aligned}$$

Since annotated or monitored programs contain more information than unannotated programs, the evaluation rules are parametrised with types *FullProgram* and *FullState*. For each instantiation we give mappings program and prog_state to the basic program type *Program* and the basic program state *PState*. Further, we add parameters that specify the actions that are taken upon method entry or (normal or exceptional) exit (γ_{IN}, γ_{NORM}, and γ_{EXC}, respectively), and the handling of annotations (δ_{SET}, δ_{ASSERT}, and δ_{CASE}). In a standard program semantics, where specifications are ignored, these are all instantiated with the identity relation.

The evaluation rules are fairly standard, and we refer to Von Oheimb and the PVS formalisation for more details. Evaluation of normally terminating method calls is described by the following rule (where for clarity of presentation, we left out several checks that intermediate states are not exceptional).

$$\frac{\begin{array}{c} \sigma_0.\mathsf{prog\_state.exc} = \bot \qquad P \vdash \langle o, \sigma_0 \rangle \triangleright \langle r, \sigma_1 \rangle \qquad P \vdash \langle p, \sigma 1 \rangle \triangleright \langle act, \sigma_2 \rangle \\ r \neq \mathsf{Null} \qquad md = \mathsf{lookup\_mthd}(P, r, mn) \\ old\_lvs = \sigma_2.\mathsf{prog\_state.store.loc\_vars} \quad \sigma_3 = \mathsf{update\_lvs}(\sigma_2, r, md.\mathsf{lvars}, md.\mathsf{param}, act) \\ \gamma_{\text{IN}}(P, md, r, \mathsf{Const}(act), \sigma_3, \sigma_4) \qquad P \vdash \langle md.body, \sigma_4 \rangle \triangleright \langle \mathbb{1}, \sigma_5 \rangle \\ P \vdash \langle md.\mathsf{res}, \sigma_5 \rangle \triangleright \langle v, \sigma_6 \rangle \qquad \gamma_{\text{NORM}}(P, md, r, \mathsf{Const}(v), \sigma_6, \sigma_7) \end{array}}{P \vdash \langle \mathsf{Call}(o, mn, p), \sigma_0 \rangle \triangleright \langle v, \sigma_7(\mathsf{prog\_state.store.loc\_vars} := old\_lvs) \rangle}$$

First the receiver is evaluated, resulting in non-null reference r. Next, the parameter is evaluated, resulting in value act. Using r, the method definition md is looked up. The local variable store is updated assigning r to `this`, initialising the method's local variables and assigning the actual parameter to the formal parameter. The old local variable store is remembered as old_lvs. Next, an appropriate action upon method entry is taken, as specified by the relation γ_{IN}. Then the method body, and method result expression are evaluated. Since this rule applies to normal method termination only, the parameter for normal method termination γ_{NORM} is evaluated. Last, the local store is set back to old_lvs. In addition, rules exist that specify behaviour of a method call when it is called upon a null reference, the body contains an uncaught exception *etc.*

Annotated Program Semantics The program state of an annotated program is extended with a store for ghost variables:

$$AState = [\#\ \mathsf{pstate} : PState, \mathsf{ghost\_vars} : \mathcal{N} \mapsto \mathcal{V}\ \#]$$

The types *FullProgram* and *Program* coincide, while *FullState* is instantiated as *AState*, and the mapping prog_state is defined as pstate. Figure 5 shows some of the instantiations of the semantics parameters; the other instantiations are similar. The relation γ_{IN} uses the auxiliary relation β which checks a boolean expression e and raises a special JMLException if it evaluates to false. Upon method entry, the class invariant and precondition are evaluated. We assume that lookup_inv returns the complete class invariant, including those invariants that are inherited from superclasses. If they fail, a JMLException is thrown, otherwise the method's pre_set statement is executed. Finally, we ensure that the program store is not changed. The function δ_{SET} updates a ghost variable: it first evaluates the expression and if this did not result in an exceptional state, it updates the value of the ghost variable[3] appropriately.

Monitored Program Semantics The parametrised program semantics is also instantiated for monitored programs. This semantics is only defined when the PA is compatible with the program. PA a is said to be compatible with a program

[3] Where $\tau(\mathsf{ghost\_vars}.n := v)$ abbreviates that the value of $\mathsf{ghost\_vars}(n)$ in τ is updated to v.

$$\frac{\begin{array}{c} inv = \mathsf{lookup\_inv}(P, r) \qquad \beta(P, inv, \sigma_1, \tau_1) \qquad \beta(P, md.\mathsf{pre}(act), \tau_1, \tau_2) \\ P \vdash \langle md.\mathsf{pre\_set}(act), \tau_1\rangle \triangleright \langle v, \tau_2\rangle \qquad v \in \{\bot, 1\!\!1\} \qquad \sigma_1.\mathsf{pstate.store} = \sigma_2.\mathsf{pstate.store} \end{array}}{\gamma_{\text{IN}}(P, md, r, act, \sigma_1, \sigma_2)}$$

$$\frac{P \vdash \langle e, \sigma_1\rangle \triangleright \langle v, \tau\rangle \qquad \mathsf{if}\ v = \mathsf{B(true)}\ \mathsf{then}\ \sigma_2 = \tau\ \mathsf{else}\ \sigma_2 = \tau(\mathsf{exc} := \mathsf{JMLException})}{\beta(P, e, \sigma_1, \sigma_2)}$$

$$\frac{P \vdash \langle e, \sigma_1\rangle \triangleright \langle v, \tau\rangle \qquad \mathsf{if}\ \tau.\mathsf{pstate.exc} = \bot\ \mathsf{then}\ \sigma_2 = \tau(\mathsf{ghost\_vars}.n := v)\ \mathsf{else}\ \sigma_2 = \tau}{\delta_{\text{SET}}(P, \mathsf{Set}(e, n), \sigma_1, \sigma_2)}$$

Fig. 5. Instantiation of semantics for runtime annotation evaluation

P, denoted $a \sqsubseteq P$, if (i) the program contains the class c that is being monitored, (ii) all variables declared as program variables in a are fields of the class c with the correct type, and (iii) every event name corresponds to a method in the class. A monitored program is a product of a PA and a program. The state of a monitored program consists of the states of the PA and the program (including ghost variables)[4], and a flag stuck. If the PA is partial, the flag stuck is set when Δ_a is not defined for a certain input. If the flag is set, this means that the security policy is violated, and the program should be stopped (by some external observer). If the PA is total, the stuck flag will never be set. Instead, violation of the security policy is modelled by the PA reaching the trap state halted (in which case the external observer again is supposed to stop execution).

$$\begin{aligned} MProgram &= [\#\ \mathsf{pa} : PA, \mathsf{program} : Program\ \#] \\ MState &= [\#\ \mathsf{pa\_state} : PAState, \mathsf{pstate} : PState, \mathsf{ghost\_vars} : \mathcal{N} \mapsto \mathcal{V}, \mathsf{stuck} : \mathbb{B}\ \#] \end{aligned}$$

Thus, *FullProgram* gets instantiated as *MProgram* and *FullState* as *MState*, with mappings program and pstate. Now we can give appropriate instantiations for the γ- and δ-relations. The δ-relations are the same as for the annotated program semantics, but the γ-relation also updates the state of the monitor. For example, γ_{IN} is defined in terms of γ_{IN} for annotated programs, as defined in Fig. 5.

$$\frac{\begin{array}{c} \gamma_{\text{IN}}^{\mathsf{AP}}(P, md, r, act, \sigma_1, \tau) \\ \mathsf{if}\ \tau.\mathsf{pstate.exc} = \bot\ \mathsf{then}\ \sigma_2 = \gamma_{\text{PA}}(\mathsf{entry})(P, md, act, \tau)\ \mathsf{else}\ \sigma_2 = \tau \end{array}}{\gamma_{\text{IN}}(P, md, r, act, \sigma_1, \sigma_2)}$$

where

$$\begin{aligned} \gamma_{\text{PA}}(ev)(P, md, act, \sigma) = \ & \mathsf{let}\ \ e = (\#\ \mathsf{etype} := ev, \mathsf{mname} := md.\mathsf{name}\ \#), \\ & \qquad \tau = \Delta_{P.\mathsf{pa}}(\sigma.\mathsf{pa\_state}, \sigma.\mathsf{prog\_state}, e, act)\ \ \mathsf{in} \\ & \mathsf{if}\ \sigma.\mathsf{stuck} \vee \tau = \bot\ \mathsf{then}\ \sigma(\mathsf{stuck} := \mathsf{true})\ \mathsf{else}\ \sigma(\mathsf{pa\_state} := \tau) \end{aligned}$$

4 Annotation Generation

Given a security property encoded as a PA, the annotation generation procedure generates JML-annotations that capture this property, *i.e.*, if the program

[4] For convenience, we assume that a monitored program also evaluates annotations, but this instantiation is in fact orthogonal to the annotated program semantics.

respects the property encoded by the monitor, then it does not violate the generated annotations. As explained above, the procedure is defined in several steps: (*i*) the monitor is completed; (*ii*) the annotations are generated at the method specification level, as special set-annotations; and (*iii*) the method specification-level set-annotations are inlined in the method body. Notice that the special CaseSet annotation could be translated into standard JML annotations as well.

For each step we prove that the new program simulates the old program, *i.e.*, we show for every translation step α there exists a relation R such that:

$$\begin{aligned}\forall b, \sigma_1, \sigma_2, \tau_1, v_1 . P \vdash \langle b, \sigma_1\rangle \triangleright \langle v_1, \sigma_2\rangle \wedge R(\sigma_1, \tau_1) \Rightarrow \\ \exists \tau_2, v_2 . \alpha(P) \vdash \langle b, \tau_1\rangle \triangleright \langle v_2, \tau_2\rangle \wedge R(\sigma_2, \tau_2)\end{aligned}$$

Additionally, we show that the initial program states are related by R, and from this we can conclude that for any reachable state of the monitored program, there exists a related state, reachable in the translated program. As a side-remark, for translation steps (*i*) and (*iii*), we can even prove that relation R is a bisimulation, while for step (*ii*) we can only prove existence of a simulation (since non-terminating monitored programs – for which no derivation exists in the natural semantics – might terminate after annotation generation, because of an annotation violation).

A natural way to prove the simulation is by induction over the derivation length. However, induction can only be applied when the body is unchanged. Since the translation introduces new (ghost) variables to encode the PA, this is not always the case. For these cases, separate preservation lemmas have to be proven. Further, to be able to complete the proof, we need to ensure that in both bodies the same branches of conditional expressions and statements are taken, and that the same values get assigned to the store. Therefore, we prove a stronger result, adding that also the values v_1 and v_2 are the same (for step (*ii*): provided the monitor did not reach the halted state).

Completion of the automaton The first translation step does not change the program itself, it only completes the PA. Suppose that P is a monitored program, where the monitor $P.\mathsf{pa}$ is deterministic and wellformed. Then the translation to a monitored program with a total PA, $\alpha_1(P)$, is defined as:

$$\alpha_1(P) = (\#\, \mathsf{pa} := \mathsf{complete}(P.\mathsf{pa}), \mathsf{program} := P.\mathsf{program}\ \#)$$

The relation that is preserved between executions of P and $\alpha_1(P)$ is the following (where σ is a state of P and τ is a state of $\alpha_1(P)$):

$$\begin{aligned}R(\sigma, \tau) = \ & (\mathsf{if}\ \sigma.\mathsf{stuck}\ \mathsf{then}\ \tau.\mathsf{pa\_state.current} = \mathsf{halted} \\ & \ \ \mathsf{else}\ \sigma.\mathsf{pa\_state.current} = \tau.\mathsf{pa\_state.current}) \wedge \neg\tau.\mathsf{stuck} \wedge \\ & (\sigma.\mathsf{pa\_state.store_A} = \tau.\mathsf{pa\_state.store_A}) \wedge \\ & (\sigma.\mathsf{pstate} = \tau.\mathsf{pstate}) \wedge (\sigma.\mathsf{ghost\_vars} = \tau.\mathsf{ghost\_vars})\end{aligned}$$

To prove that this relation is preserved for any body b, we use equivalence (1) on Page 4 and we observe further that (*i*) if stuck has been set, it remains set, (*ii*) for a total PA, if halted is reached, it is never left, and (*iii*) for a total

$$\begin{aligned}
\alpha_2(P) &= \texttt{(\#}\ \mathsf{classes} := \{\alpha_{2,\mathcal{C}}(c, P.\mathsf{pa}) \mid c \in P.\mathsf{program.classes}\}\ \texttt{\#)} \\
\alpha_{2,\mathcal{C}}(c, a) &= \mathsf{if}\ c.\mathsf{name} \neq a.\mathsf{clname}\ \mathsf{then}\ c \\
&\qquad \mathsf{else}\ c\ \texttt{(\#}\ \mathsf{ghost\_vars} := c.\mathsf{ghost\_vars} \cup \mathsf{new\_vars}(a) \\
&\qquad\qquad\qquad \mathsf{inv} := \mathsf{And}(\mathsf{Not}(\mathsf{Eq}(\mathsf{cp}, \texttt{halted})), c.\mathsf{inv}) \\
&\qquad\qquad\qquad \mathsf{methods} := \{\alpha_{2,\mathcal{M}}(m, a) \mid m \in c.\mathsf{methods}\}\ \texttt{\#)} \\
\alpha_{2,\mathcal{M}}(m, a) &= m\ \texttt{(\#}\ \mathsf{pre\_set} :=\ m.\mathsf{pre\_set}; \alpha_{2,\mathcal{E}}(\mathsf{entry}, m.\mathsf{name}, a); \\
&\qquad\qquad\qquad\qquad \mathsf{Assert}(\mathsf{Not}(\mathsf{Eq}(\mathsf{cp}, \texttt{halted}))), \\
&\qquad\qquad \mathsf{post\_set} := m.\mathsf{post\_set}; \alpha_{2,\mathcal{E}}(\mathsf{exit}, m.\mathsf{name}, a) \\
&\qquad\qquad \mathsf{exc\_set} := m.\mathsf{exc\_set}; \alpha_{2,\mathcal{E}}(\mathsf{exc\_exit}, m.\mathsf{name}, a)\ \texttt{\#)} \\
\alpha_{2,\mathcal{E}}(e, n, a) &= \alpha_{2,\mathcal{T}}(\{t \mid t \in a.\mathsf{trans} \wedge t.\mathsf{etype} = \texttt{(\#}\ \mathsf{event} := e, \mathsf{mname} := m\ \texttt{\#)}\}, a) \\
\alpha_{2,\mathcal{T}}(ts, a) &= \mathsf{CaseSet}(\{(\mathsf{Eq}(\mathsf{cp}, \mathsf{q}), \alpha_{2,\mathcal{S}}(ts, q)) \mid \mathsf{q} \in a.\mathsf{cps}\}) \\
\alpha_{2,\mathcal{S}}(ts, q) &= \mathsf{CaseSet}(\{(t.\mathsf{guard}, \mathsf{Set}(\mathsf{cp}, t.\mathsf{dest}; t.\mathsf{action})) \mid t \in ts \wedge t.\mathsf{source} = \mathsf{q}\})
\end{aligned}$$

Fig. 6. Formal definition of translation PA into annotations

PA, stuck is never set. Formally, where P is a monitored program, and Q is a monitored program with total PA:

$$\begin{gathered}
\sigma_1.\mathsf{stuck} \wedge P \vdash \langle b, \sigma_1 \rangle \triangleright \langle v, \sigma_2 \rangle \Rightarrow \sigma_2.\mathsf{stuck} \\
\sigma_1.\mathsf{pa\_state.current} = \mathsf{halted} \wedge Q \vdash \langle b, \sigma_1 \rangle \triangleright \langle v, \sigma_2 \rangle \Rightarrow \sigma_2.\mathsf{pa\_state.current} = \mathsf{halted} \\
\neg\sigma_1.\mathsf{stuck} \wedge Q \vdash \langle b, \sigma_1 \rangle \triangleright \langle v, \sigma_2 \rangle \Rightarrow \neg\sigma_2.\mathsf{stuck}
\end{gathered}$$

To illustrate how the annotation generation algorithm works on the LimitSMS automaton in Fig. 1, assume we have declared a class `Messaging`, containing the methods used by the automaton, plus a method `receiveSMS`. Applying translation α_1 means that this class, instead of being monitored by the partial PA in Fig. 1, is monitored by the total PA in Fig. 2.

From PA to Annotations Figure 6 contains the formal definition of the second translation step: from PA to method-level set-annotations. Given a monitored program P where P.pa is total, the annotation generation algorithm α_2 applies $\alpha_{2,\mathcal{C}}$ to all classes. This function checks whether the class is the one being monitored. If so, appropriate ghost variables are added to the class using the function new_vars (see the PVS formalisation for its formal definition). Basically (*i*) for each automaton control point q, a (final) ghost variable declaration q is generated, initialised to a unique value; (*ii*) a ghost variable cp is declared, initialised to the value of the ghost variable representing the initial control point; and (*iii*) for each automaton variable declaration, a ghost variable is declared with corresponding name, type and initialisation. Further, $\alpha_{2,\mathcal{C}}$ adds the condition that the current control point should not be halted to the class invariant[5], and it annotates all methods in the class using $\alpha_{2,\mathcal{M}}$. For each method, pre_set, post_set and exc_set are extended with updates to the ghost variables encoding the automaton. In addition, at the end of pre_set, an Assert statement is added

[5] For readability, we do not explicitly write the translation from PA control points to ghost variables.

to verify whether the transition reached the halted state: in that case program execution should terminate immediately. Without this Assert, the property violation would only be detected after the body is executed. To encode the updates to the ghost variables, $\alpha_{2,\mathcal{E}}$ computes the set of relevant transitions (*i.e.*, those where the event and method name correspond). For these transitions, a CaseSet statement is generated, where the different cases correspond to the current control point being equal to a control point q, for any q in the automaton. For each such q, $\alpha_{2,\mathcal{S}}$ selects the transitions where t.source is q and generates a CaseSet statement, that tests whether the guard holds, and if so, sets the control point cp to t.dest, and executes the actions associated with this transition. Notice that the order in which the different cases are generated is not important: since the PA is total and deterministic there is always exactly one case that applies.

The formalisation does not specify how guards and actions are translated. Instead, we assume there exists a translation into expressions in the programming language that (*i*) are wellformed, (*ii*) give the same result, (*iii*) do not have side-effects, (*iv*) do not throw exceptions, and (*v*) do not contain method calls. From this we can derive that in the annotated program, the generated statements in pre_set can only throw a JMLException (because of the concluding Assert), while the generated statements in post_set and exc_set do not throw any exception.

As an example, consider again the class `Messaging` and the completed PA, encoding the *limited SMS* policy, in Fig. 2. Figure 7 shows the generated annotations that result from applying translation $\alpha_{2,\mathcal{C}}$ on this class and this PA. Notice that for methods and events that are not involved in the property, an empty CaseSet is generated – this is equivalent to a Skip statement.

To show correctness of the translation, we show that the following relation is preserved (where P is the monitored program, σ a state of the monitored program, and τ a state of the annotated program):

$$\begin{array}{rl}
R(\sigma,\tau) = & \neg\sigma.\mathsf{stuck} \wedge \\
& \mathsf{if}\ \sigma.\mathsf{pa\_state.current} = \mathsf{halted}\ \mathsf{then}\ \tau.\mathsf{pstate.exc} = \mathsf{JMLException}\ \mathsf{else}\ S(\sigma,\tau)\\
S(\sigma,\tau) = & (\mathsf{unique}(\sigma.\mathsf{pa\_state.current}) = \tau.\mathsf{ghost\_vars}(\mathsf{cp})) \wedge \\
& \forall q \in P.\mathsf{pa.cps.}\ (\mathsf{unique}(q) = \tau.\mathsf{ghost\_vars}(\mathsf{q})) \wedge \\
& \forall n \in \mathcal{N}.(\sigma.\mathsf{pa\_state}(n) \neq \perp \Rightarrow \sigma.\mathsf{pa\_state}(n) = \tau.\mathsf{ghost\_vars}(n)) \wedge \\
& \sigma.\mathsf{pstate} = \tau.\mathsf{pstate} \wedge \\
& \forall n \in \mathcal{N}.(\sigma.\mathsf{ghost\_vars}(n) \neq \perp \Rightarrow \sigma.\mathsf{ghost\_vars}(n) = \tau.\mathsf{ghost\_vars}(n))
\end{array}$$

This relation specifies that if the monitor has reached control point halted, the annotated program must have thrown a JMLException. Otherwise, the state of the annotated program corresponds to the state of the original program, extended with the modelling of the monitor's state. This means that the program states (fields, local variables and exceptions) have to coincide, just as the values of the ghost variables that are declared in the original program P. Further, the current control point is represented by the value stored in ghost variable cp, and all PA control points and variables correspond to ghost variables. Notice that if an annotation already present in P causes a JMLException, both the monitored and the annotated program will throw it. Therefore, we cannot prove that the annotated program throws a JMLException *if and only if* halted is reached.

```
class Messaging {
//@ ghost int halted = 0, s1 = 1, s2 = 2, N = 5, cp = s1, n = 0;
//@ public invariant cp != halted;
/*@ pre_set  CaseSet [(cp == s1, CaseSet [(n < N, cp = s2),
                                          (n >= N, cp = halted)]),
                      (cp == s2, CaseSet [(true, cp = halted)]),
                      (cp == halted, CaseSet [(true, cp = halted)])];
             Assert cp != halted;
    post_set CaseSet [(cp == s1, CaseSet [(true, cp = halted)]),
                      (cp == s2, CaseSet [(true, cp = s1; n = n + 1)]),
                      (cp == halted, CaseSet [(true, cp = halted)])];
    exc_set  CaseSet [(cp == s1, CaseSet [(true, cp = halted)]),
                      (cp == s2, CaseSet [(true, cp = s1)]),
                      (cp == halted, CaseSet [(true, cp = halted)])]; @*/
void sendSMS(){/* body sendSMS */}
/*@ pre_set  CaseSet []; Assert cp != halted;
    post_set CaseSet []; exc_set  CaseSet []; @*/
void receiveSMS(){/* body receiveSMS */}
/*@ pre_set  CaseSet []; Assert cp != halted; exc_set CaseSet [];
    post_set CaseSet [(cp == s1, CaseSet [(true, cp = s1; n = 0)]),
                      (cp == s2, CaseSet [(true, cp = halted)]),
                      (cp == halted, CaseSet [(true, cp = halted)])]; @*/
void reset() {/* body reset */} }
```

Fig. 7. Method-level set annotations generated for class Messaging

To prove that this relation is a preserved, it is strengthened with the following property: if the control point is not halted, then the derivations also produce the same value. The crucial part in the proof is of course what happens upon method call and termination. For example, when a method is called, first the invariant and the precondition are evaluated. Assuming that halted is not yet reached, the new conjunct of the invariant evaluates to true, and induction allows to derive that after evaluation of the precondition, the states are related by R. Next, the original pre_set annotations are evaluated, and again the induction hypothesis allows to conclude that the resulting states are related. Next, the monitored program makes a PA transition, and the annotated program executes the newly generated set annotations, followed by an Assert to check whether halted has been reached. Here we cannot use the induction hypothesis, but instead we show manually that relation R is preserved. Notice that in post_set or exc_set we do not have an Assert statement. Since the invariant is evaluated immediately after the set-annotations, the reaching of halted will be detected immediately. For this part of the proof it is crucial that the newly added invariant is evaluated first.

Finally, to complete the proof, we have to add a restriction to programs. We follow the Java Language Specification in describing its behaviour [6]. This means in particular that if the *finally* block in the statement terminates abnormally (because of an exception, or any other reason for abrupt completion), it overrides a possible exception thrown in the *try* or *catch* block. Thus, for example, if halted

$$
\begin{aligned}
\alpha_{3,\mathcal{M}}(P,m) = m(\#\ & \mathsf{pre\_set} := \mathsf{Skip}, \mathsf{post\_set} := \mathsf{Skip}, \mathsf{exc\_set} := \mathsf{Skip},\\
& \mathsf{lvars} := \{\mathsf{result}\} \cup m.\mathsf{lvars}, \mathsf{res} := \mathsf{lookup}(\mathsf{result}),\\
& \mathsf{body} := \mathsf{TryCatchFinally}(\\
& \qquad \mathsf{TryCatchFinally}(\, m.\mathsf{pre\_set}; m.\mathsf{body};\\
& \qquad\qquad\qquad\qquad \mathsf{Assign}(\mathsf{result}, m.\mathsf{res}); m.\mathsf{post\_set}\\
& \qquad \mathsf{Throwable}, m.\mathsf{exc\_set}, \mathsf{Skip}),\\
& \quad \mathsf{RunTimeException}, m.\mathsf{exc\_set}, \mathsf{Skip})\ \#)
\end{aligned}
$$

Fig. 8. Formal definition of annotation inlining for methods

is reached in the *try* block, and hence a JMLException is thrown, this exception might be overwritten by an exception thrown in the *finally* block (see also [8] for a discussion of this problem), which would mean that the violation of the security policy is not signalled to the user, and instead execution continues (with another exception). To avoid this, for all TryCatchFinally statements in the program, we require that if the *try* or *catch* block throws a JMLException, the whole statement also terminates exceptionally because of a JMLException.

Inlining the Annotations Once the set-annotations at method specification level are generated, the next step is to inline them into the method bodies. To ensure that the appropriate set-statements are always executed at the end of the method body, the body is wrapped in a TryCatchFinally statement. The translation α_3 applies $\alpha_{3,\mathcal{C}}$ to all classes, which in turn applies $\alpha_{3,\mathcal{M}}$ to all methods in the class. This function generates one new local variable[6] result. The body of the method is changed as follows: all code is wrapped in two TryCatchFinally statements, to catch Throwable and RunTimeException exceptions[7]. In the *try* block, first pre_set is executed, followed by the body of the method. Then the result expression from the original body is evaluated, and assigned to result. Next, post_set is executed. Notice that the latter is only executed if the body actually terminates normally, otherwise the exception will simply be propagated. Finally, in the *catch* clauses, exc_set is executed. The new result expression of the method is the look up of the variable result. To conclude, pre_set, post_set and exc_set in the method specification are set to Skip. Figure 8 gives the formal definition of $\alpha_{3,\mathcal{M}}$ (where P is a program, and m a method).

To prove correctness of this translation, we use the following relation: all fields and ghost variables coincide, exceptions coincide, and all local variables that are declared in the original program coincide. In the correctness proof, we use that the post_set and exc_set annotations do not throw any exceptions, and pre_set may only throw a JMLException. Moreover, we use that the set-annotations do not contain method calls, from which we can conclude that they do not modify

[6] In fact, this should be a local *ghost* variable, but these are not yet supported by our formalisation, therefore we formalise it as a standard local variable.

[7] For simplicity, we do not model the exception hierarchy and thus TryCatchFinally can only catch a single exception, but in practice only one *try-catch-finally* instruction would be necessary.

any variables that are not explicitly mentioned in them. In particular, this allows to conclude that the new local variable is not changed by the set annotations.

5 Related Work

Security automata [13] are widely used for monitoring security properties. The originality of our work lies in considering them as specifications in a general specification language, with the ultimate goal of static verification.

Closely related to our approach is work by Aktug *et al.* [3, 1, 2], who define a formal language for security policy specifications, ConSpec, that is similar to our PAs. They prove that a monitor can be inlined into the program's bytecode, by adding first-order logic annotations, and then they use a weakest precondition computation that essentially works the same as the annotation propagation algorithm that we plan to use [12] to produce a fully annotated, verifiable program. In contrast, our algorithm is defined for source code, and connects with the general-purpose specification language JML. This allows the use of JML verification tools, to verify the actual policy adherence. And of course, correctness of our inlining algorithm has been proven with a theorem prover.

Cheon and Perumendla propose an extension of JML to specify allowed sequences of methods calls in a regular expression-like notation [4]. This results in succinct specifications, but of limited expressiveness. Even our Limited SMS example is out of their scope, because it contains a counter used only by the specification. Further, they only target runtime verification.

Several tools exist that translate temporal properties into JML annotations: AutoJML [7] translates finite state machine specifications into JML annotations and can also generate a code skeleton for a smart card applet; JAG [5] translates properties in (a subset of) temporal logic, including liveness properties. However, they typically do not distinguish between method entry and exit, and moreover, correctness of the translation algorithm has not been proven.

For more information about policy languages, monitor inlining and specifying policy adherence, we refer to Section 4.10 of Aktug's thesis [1].

6 Conclusions & Future Work

This paper presents an algorithm to inline security automata, in the form of JML annotations. The translation is defined in several steps, thanks to the introduction of method-level set-annotations as extension to JML. All steps are formalised and proven correct, using the PVS theorem prover. The algorithm might seem trivial, but several subtleties complicate the proof, *i.e.* evaluating the specifications in the right order, dealing with side-effect-freeness of annotations and the possibility that a *finally* block hides exceptions.

The formalisation has been developed for a subset of Java. We believe that extending it to full (sequential) Java would be relatively straightforward. However, generalising to properties that are not restricted to a single class or that are related to multithreading might be more challenging.

The ultimate goal of our work is to statically verify adherence to security policies. To achieve this, a weakest precondition calculus can be used to generate pre- and postconditions, based on the generated Set annotations. In earlier work, we presented such a propagation algorithm [12], and proved correctness for a limited case (instance variables and branches are not considered). It is future work to overcome these limitations.

References

1. I. Aktug. *Algorithmic Verification Techniques for Mobile Code*. PhD thesis, Royal Institute of Technology (KTH), Sweden, 2008.
2. I. Aktug, M. Dam, and D. Gurov. Provably correct runtime monitoring. In *Formal Methods (FM'08)*, volume 5014 of *LNCS*, pages 262–277, 2008.
3. I. Aktug and K. Naliuka. ConSpec: A Formal Language for Policy Specification. In *Run Time Enforcement for Mobile and Distributed Systems (REM'07)*, volume 197-1 of *Electronic Notes in Theoretical Computer Science*, pages 45–58, 2007.
4. Y. Cheon and A. Perumendla. Specifying and Checking Method Call Sequences of Java Programs. *Software Quality Journal*, 15:7–25, 2007.
5. A. Giorgetti and J. Groslambert. JAG: JML Annotation Generation for verifying temporal properties. In *Fundamental Approaches to Software Engineering (FASE 2006)*, volume 3922 of *LNCS*, pages 373–376. Springer, March 2006.
6. J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification, third edition*. The Java Series. Addison-Wesley, 2005.
7. E. Hubbers, M. Oostdijk, and E. Poll. From finite state machines to provably correct Java Card applets. In D. Gritzalis, S. De Capitani di Vimercati, P. Samarati, and S.K. Katsikas, editors, *IFIP Information Security Conference*, pages 465–470. Kluwer Academic Publishers, 2003. See `http://autojml.sourceforge.net`.
8. M. Huisman. Run-time verification can miss errors - why finally clauses can be dangerous, 2008. Manuscript.
9. G. T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. Cok, and J. Kiniry. *JML Reference Manual*, July 2005. In Progress. Department of Computer Science, Iowa State University. Available from `http://www.jmlspecs.org`.
10. D. von Oheimb. *Analyzing Java in Isabelle/HOL: Formalization, Type Safety and Hoare Logic*. PhD thesis, Technische Universität München, 2001.
11. S. Owre, S. Rajan, J.M. Rushby, N. Shankar, and M.K. Srivas. PVS: Combining specification, proof checking, and model checking. In R. Alur and T.A. Henzinger, editors, *Computer-Aided Verification (CAV '96)*, volume 1102 of *LNCS*, pages 411–414. Springer, 1996.
12. 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.A. El Kalam, editors, *Cardis'04*, pages 1–16. Kluwer, 2004.
13. F.B. Schneider. Enforceable security policies. Technical Report TR99-1759, Cornell University, October 1999.

Acknowledgements We thank Erik Poll for his useful comments on an earlier draft of this paper, and Igor Siveroni, who started the work on this topic and came up with the idea to use method-level set-annotations.