Verifying Security Policies by using JML annotations

Nils Klatter, University of Twente, Netherlands

August 7, 2011

Abstract

Security policies are an intuitive way to express security properties of a program and therefore easy to use. Usually security policies are expressed as security automata and used by a monitor, which checks the compliance of the program to the security policy at run time. If the program violates the security policy, the program is halted. Because this behavior is not always desirable, Huisman and Tamalet introduced a method to check the adherence of Java programs to a security policy statically without executing the program. The method translates a security automaton into JML annotations that inline the monitor. However, this method only supports static Java classes. Concretely, this assignment extends the supported Java language to support non-static classes and methods. Furthermore, it adapts the notation of the security automaton such that an automaton transition can be associated to the execution of a method of a non-static classes. This enables the monitoring of non-static classes with a single monitor. The assignment also adapts the translation method of Huisman and Tamalet to inline the extended monitor in the Java code.

Contents

List of Figures

Chapter 1

Introduction

The increase of internet-enabled wireless devices such as a Personal Digital Assistants (PDAs) and laptops puts a greater emphasis on program security [4]. A way to express the desired security properties of a program is via security policies. A security policy describes the allowed run-time behavior of a program [11]. Two examples of desired properties described as security policies are that a program on a PDA should not be allowed to gain unauthorized access to its location services and the PDA owner does not want an application to send his contact list over the Internet.

There are different techniques available to enforce such a security policy. For example, via a third party (i.e. TRUSTe.com [12]) issuing certificates to the developers of a program. However, certification procedures are costly and time consuming and therefore not well suited for all applications.
A different approach is Execution Monitoring (EM). This approach monitors the execution of a program at run time and halts the program if the security policy is violated [11].
Another approach is static verification. As opposed to EM, this approach verifies if the program adheres to the security policy without executing the program.

The remainder of this chapter is organized as follows. Section 1.1 explains EM in more detail. Section 1.2 gives an overview of static verification. Section 1.3 introduces the research goals of this assignment. Last, Section 1.4 gives an overview of the rest of the chapters.

1.1 Execution monitoring

An EM monitors the execution of a program with a Security Automaton (SA). The SA represents a security policy and consists of states and transitions labeled with the entry or exit of monitored methods of the program. When the event of entering or exiting a monitored method occurs, the internal state of a SA is updated by taking a transition representing the event. The encoding is such that if the SA reaches an illegal state, the security policy is violated by the program and the EM will stop the execution of the program [7].

1.1.1 Example of a security automaton

Consider a situation where a customer swipes his cash card through a payment terminal and enters his pincode afterwards. A transaction is only approved if the entered pin code matches the pin code stored at the bank which issued the card. To prevent abuse, the bank enforces a security policy which states that a customer has a maximum of 3 consecutive attempts to enter a correct pincode before his card is blocked.

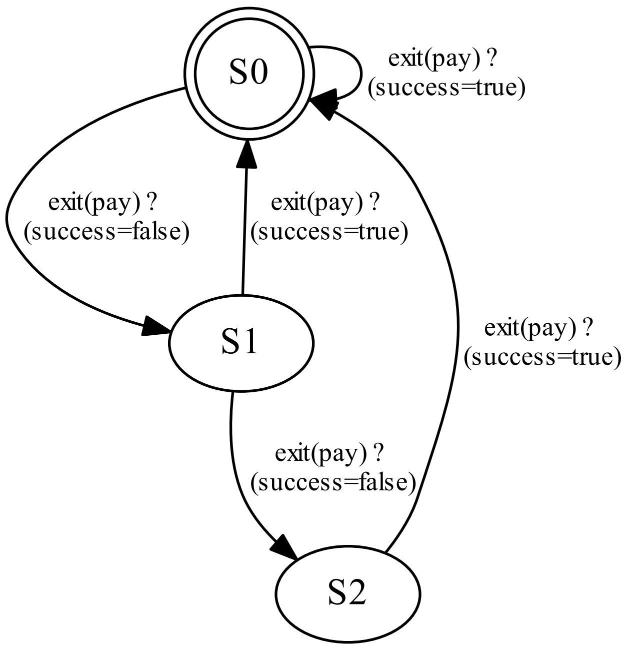

Figure 1.1: Simplified property automaton: maximum 3 wrong pin

The SA representing the bank's security policy is shown in Figure 1.1. The SA contains three states, corresponding to the number of incorrect attempts made by the customer. In s0, no incorrect attempts have been made, whereas in s1 and s2 one or two incorrect attempts have been made, respectively. Furthermore, if a client enters his pincode correctly within the maximum of 3 tries, it returns to state s0.

The behavior of terminal approving or rejecting the transaction is shown in Figure 1.2. From the program's perspective, a customer can initiate a transaction by invoking the try method and pass his pincode as an argument. Next, the program verifies the pincode and returns the success or failure of the attempt via the success variable. The transitions of the SA of Figure 1.1 are all triggered by the 'try method' of the monitored program of Figure 1.2. When the evaluation of this method has finished, the EM checks if the attempt was successful or a failure. Depending on the outcome, the EM updates the state of the SA accordingly.

```
public class Terminal {
...
int pincode = ...;\\\\ the pincode
int success = ...;\\\\ true if the try method accepted the pincode
...
public boolean int try(int pincode) {
... \\\\check if the pin code is correct
return success;
}
```

Figure 1.2: Source code monitored Java program

1.1.2 Execution monitoring as a testing technique

An alternative of Execution Monitoring is applying it as a testing technique. According to Myers: "Software Testing is the process of executing a program or system with the intent of finding errors" [3]. When a program is halted by an EM, it indicates that a security policy has been violated. This property can be used to find errors in a program which violate the security policy and thereby testing the program. When a violation is detected, the developer can use the trace of the execution and the program state to correct the error. Furthermore, the goal of testing is to design test runs such that they ideally find all of the errors. For a more in depth explanation about different kind of testing techniques, see Pan [9].

1.1.3 Advantages and disadvantages of execution monitoring

The advantage of Execution Monitoring (EM) is that security policies can be quickly and cheaply implemented by a programmer when there is an existing EM tool available. In this case, the programmer only has to create a Security Automaton (SA) based upon the security policy. Furthermore, the EM is independent of the monitored program, so new SAs can be added to or removed from the EM without affecting the source code of the monitored program. Thus a SA can be created without the involvement of the developer of monitored program.
However, the disadvantage of EM is that the program halts after a violation is detected. Thus for programs requiring high availability, this method is not well suited. Furthermore, this method is not user friendly. If a violation is detected, it simply forces the user to restart or stop using the program. To prevent such

situations, the developer can use EM as a testing techniques as explain in the previous section. However, testing can be time consuming and therefore cost intensive.

1.2 Static verification

An alternative to Execution Monitoring is static verification. The main difference between EM and static verification is, that static verification does not involve the execution of the program. Static verification verifies the formal specification of the program by analyzing the source code. Furthermore, static verification exactly verifies the adherence of a program to its specification and thereby is an exact method. However, creating the formal specification and the verification thereof is more complex than execution monitoring.

To counter the disadvantages of EM and use the advantages of static verification, Huisman and Tamalet suggest that adherence of a program to its security policies can be verified via static verification automatically [7]. The approach of Huisman and Tamalet tries to achieve the goal of automated static verification of security policies by expressing a security policy in the form of JML annotations along with the Java source code.

JML is a Behavioral Interface Specification Language (BISL) which can be used to formally specify the behavior and interfaces of Java classes and methods [8]. It uses the Design by Contract (DBC) principle where the specification of a class can be considered a contract. The client using the class is guaranteed to get certain results (invariant and postconditions) from a class method when it adheres to the given requirements of the class and the method. (invariants and preconditions)

The Java language is a platform independent high level programming language, which supports advanced features such as concurrency, objects, overriding and inheritance [5]. Furthermore, the Java language is platform independent, such that Java programs are runnable on multiple platforms with the same source code.

A Java program is organized in a hierarchy of interfaces and classes allowing the programmer to use an 'object-oriented' approach. The Java language considered in Chapter 2 is a simplified version of the original Java programming language leaving out the high level features of concurrency, inheritance and overriding.

1.2.1 Example of a JML annotated Java program

Figure 1.3 shows an example of the JML annotated source code of Figure 1.2. The JML annotations are enclosed in /*@ .. @*/ and are ignored by the regular Java compiler. A static verification tool however, uses the JML annotations to check the conformance of its implementation. In this example, the invariant of the pincode variable can take a value from 0 up until 9999. Furthermore, the JML annotations at the try method forces the result only to be true if the parameter pin is equal to the retrieved pincode.

```
public class Terminal {
...
/*@ public invariant pincode > 0 && pincode < 10000; @*/
int pincode = ...;\\\\ the pincode retrieved from an external location
int success = ...;\\\\ true if the try method accepted the pincode
...
/* @ ensures result == (pin == pincode)  @*/
public boolean int try(int pin) {
... \\\\check if the pin code is correct
return success;
}
```

Figure 1.3: JML annotation Java program

1.2.2 Verifying security policies using static verification

Security policies are most conveniently expressed in the form of a Security Automata (SA). However, the method of Huisman and Tamalet performs static verification based on JML annotated Java code. Therefore, a translation method is needed, which translates a SA into JML annotations for a Java program. In the previous example, the JML annotations have been added manually and are not related to the security policy as defined in 1.1. In order to simplify the process of static verification, the method of Huisman and Tamalet performs this translation automatically.

The class of Security Automata which can be handled by Huisman and Tamalet's translation function is limited to a single Security Automaton monitoring a single class. Also the set of instructions of the Java language supported is simplified. For an exact syntax definition see Chapter 2.

This assignment extends the method of Huisman and Tamalet such that a single Security Automaton can monitor multiple instances of a set of classes. Also the syntax of the simplified Java language is extended with for example the 'new' statement creating a new object instance and the static/non-static modifier in the variable declaration [7].

When monitoring multiple objects one has to extend the notation of the existing SA definition and also define a way to annotate these objects. The solutions to these problems are given in Chapter 4 by extending the Security Automaton.

The extended translation method of Chapter 6 is divided into three stages. The first stage completes the PA by adding the behavior which violates the security policy. The second stage generates the JML annotation at the beginning of each method, describing an equivalent behavior as the completed automaton. The third stage wraps the JML annotation at the beginning of each method in the method body.

After the last stage in principle, the generated JML annotations does not describe the complete behavior of the Java program, but only of the security policy. Therefore, it is not directly compatible with static verification. However, one can achieve automatic static verification by generating pre and post conditions based upon the set annotations via weakest precondition calculus. Such a method is described in a limited case by Pavlova et al [10]. Nonetheless, manually annotating the Java code remains time consuming. Thus to reach the goal of fully automated verification of security policies, one has to further extend the translation method.

1.3 Research Goals of this assignment

Main goal:
How can the existing security policy translation method of Huisman and Tamalet be extended to support multipe object instances? [7]

Sub goals:

- How does this affect the syntax of the simplified Java language?
- What are security policies and security automata?
- How is a property automaton related to a security automaton and how is the property automaton affected?
- How are the operational semantics of the simplified Java language affected?
- How is the translation method of Huisman and Tamalet affected?

1.4 Overview of this assignment

This assignment is organized as follows.

- Chapter 2 gives the syntax of the new simplified Java language and discusses the differences with the simplified Java language of Huisman and Tamalet; [7]
- Chapter 3 explains the concepts of security policies and security automaton via an example;
- Chapter 4 gives the formalization of the property automaton, which is derived from the security automaton of Chapter 3 and Huisman and Tamalet; [7]
- Chapter 5 gives an overview of the relevant operational semantics of the simplified Java language of Chapter 2;

- Chapter 6 formalizes the translation from a property automaton into inlined JML annotations;

- The conclusion revisits the research goals and suggests possible future work.

Chapter 2

Syntax

This chapter gives an overview of a simplified subset of the Java programming language. It abstracts away from advanced language features such as inheritance, overriding and concurrency. Besides the regular Java syntax, the Java methods can contain JML annotations supporting the monitoring of the Security Policy.

The Java syntax shown in Figures 2.1, 2.2 and 2.3 is an extended version of the one defined by Huisman and Tamalet [7]. The original syntax definition is shown in regular font and the extensions are bold faced. The next sections will review each syntax definition line by line and the last section introduces the program state.

2.1 Value, type, exceptions and declarations

The data structure of a Java program is defined by its variables and its syntax is shown in Figure 2.1.

The *Decl* statement declares a variable which can be a field, a method parameter, a local variable or a ghost variable.
Each declaration consists of the following parts:

i) staticMod is a modifier of type Bool, which indicates if the variable is static or non-static

ii) type is the type of the variable

iii) name is the name of the variable

iv) init is the initial value of the variable.

The only extension made to the syntax definition of Huisman and Tamalet is the staticMod property [7]. When staticMod is true (static), it implies that the variable should be initialized only once and should persist during the lifespan of its corresponding class. When staticMod is false (non-static), it implies that multiple copies of the variable may exists.

The set of possible Java types and values are:

i) Bool which has a corresponding boolean value of either false or true;

ii) Int which in the formalization can take any integer value, but in a runtime environment it will be limited by its memory size;

iii) Ref which is a pointer to an object and has a value which is either Null or a number representing te location where the object is stored;

iv) Void with value $\mathbb{1}$, modeling a method without a result;

v) $\perp$ denotes an undefined value and has no type. For example, when a method in a Java program throws an exception, the method returns a value which is undefined.

$$\begin{array}{rcl}
\mathcal{V} & = & \mathsf{B}(b : \mathbb{B}) \mid \mathsf{I}(i : \mathbb{Z}) \mid \mathsf{Null} \mid \mathsf{R}(i : \mathbb{Z}) \mid \mathbb{1} \mid \perp \\
\mathcal{T} & = & \mathsf{Bool} \mid \mathsf{Int} \mid \mathsf{Ref} \mid \mathsf{Void} \\
\mathcal{E} & = & \mathsf{Throwable} \mid \mathsf{RunTimeException} \mid \mathsf{JMLException} \\
\mathit{Decl} & = \texttt{[\#} & \mathsf{staticMod} : \mathbb{B}, \mathsf{type} : \mathcal{T}, \mathsf{name} : \mathcal{N}, \mathsf{init} : \mathcal{V}\ \texttt{\#]}
\end{array}$$

Figure 2.1: Syntax of Java value, type, exceptions and declaration

Except for the exceptions (RunTimeException and JMLException), all the possible errors and exceptions thrown by Java are represented by its root class Throwable. The RunTimeException exception represents the case when the Java program fails an operation during run-time and the JMLException occurs when the specified JML behavior specification is violated.
Note however, that an exception does not necessarily have to violate the accepted behavior of a program. For example, if a method throws an exception when a user enters invalid input, Java gives the programmer the opportunity to catch the exception and handle it accordingly. In general, an exception is allowed when it is specified in the JML annotations of the Java program.

In conclusion, this section defined which kinds of data variables can be declared and introduced the concept of exceptions. The next section elaborates on how the declared variables can be evaluated and manipulated.

2.2 Expression and statement

This section gives the syntax definitions of a Java expression and statement, shown in Figure 2.2. The main difference is that an expression returns a value and a statement does not. (or discards the result)

The Plus expression represents the addition of two other expressions, which should evaluate into a value of the set $\mathbb{Z}$. Note, that the left out minus and multiplication operator that are left out can be expressed as an expression with plus operators and is closed under $\mathbb{Z}$. This is not applicable for the division and the modulo operator and therefore these are not supported by this simplified Java language.

The Not, And, Eq are boolean operators whose results are booleans of type $\mathbb{B}$. The Not and And operators only take expressions with a result type boolean, while Eq can handle expressions of type I and $\mathbb{B}$.

The Varexpressions look up a variable with name $\mathcal{N}$ and of type $\mathbb{B}$, I or R, respectively. If the receiver reference is not Null, the Varexpression refers to a static/non-static ghost or field variables of the receiver object. If the receiver reference is Null, the Varexpression is a local or a method parameter variable. For a detailed description of the variable lookup semantics, see Section 5.2.

The Call expression represents a method call of method named mn of object o and with parameter p. After the execution of the call the corresponding method result is returned as the expression result. For a detailed description and its operational semantics, see Section 5.1.

The CondExpr expression is similar to the IfThenElse statement. First, the conditional expression c is evaluated to a Bool true or false. If true, expression e_1 is executed. Otherwise expression e_2 is evaluated. Last, the result of CondExpr expression is the evaluated expression, which is the result of either e_1 or e_2.

The Const expression is a value of type $\mathcal{V}$. The value is determined during its initialization and does not change during the lifetime of its existence. The Assign expression evaluates the expression e and stores it in a variable named n with receiver t. For a detailed description and its operational semantics, see Section 5.5.

Next, the New expression creates a new object instance of Class cname and passes parameter p to the constructor of the object. For a detailed description and its operational semantics, see Section 5.3.

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

Figure 2.2: Syntax of Java expression and statement

The Skip statement represents the case when no action is performed. Furthermore, the Sequence, IfThenElse and While statement combines, selects or repeats given statements in a self explanatory manner and the StmtExpr evaluates the expression, but discards the resulting value.

During the translation, the JML annotations are generated for each method based upon the completed automaton. These JML annotations can also contain variables, which are exclusively used and declared in the JML annotation and not in the Java code itself. These variables are called ghost variables and can be manipulated using the Set statement.

When an annotated Java method is evaluated, the JML annotation first determines the current automaton state. Next, it verifies which transition from the current automaton state is taken. The CaseSet statement helps the JML annotated code to select the automaton state and transition.
A CaseSet statement consists of a list of expressions and statements and is comparable to the Java case statement. First, the expressions of the list are evaluated until an expression holds true. Next, the statement corresponding to the expression is executed. Figure 6.5 shows the CaseSet constructs of the running example. For a detailed description, see Section 6.3.

The Throw and TryCatchFinally statements define the exception handling behavior of a Java program, but they are also used in the last stage of the translation method. In the last stage of the translation method, the CaseSet notation of the previous paragraph are in-lined into the method body. When a security policy is violated at a certain point in the code, a JMLException is thrown and handled by the surrounding TryCatchFinally statement. For a more detailed explanation, see Section 6.5.

In conclusion, this section shows how data in Java program should be evaluated and manipulated and how these statements and expressions should be structured. The next section structures these statements at an even higher level into method, classes and a program.

2.3 Method, class and program

The concepts of methods, classes and a program are at the heart of Java programming language and is part of the Object Oriented Programming paradigm. (OOP) In the OOP paradigm a program consists of a set of objects, which has a set of variables and methods. The objects are self-contained entities, which can interact with each other by calling each others methods or manipulating the object variables. This interaction between the self contained objects, allows a programmer to use more flexible design patterns [5].

In the simplified Java language, a program consists of a set of classes and the set of classes consists of field variables and methods. The syntax definition is shown in Figure 2.3 and will be explained in the order of *Method*, *Class* and *Program*.

The method definition represents a non-static class method. It has a name, accepts a single parameter, has a method body and returns an expression of a certain type. (including void)

Furthermore, the method's JML annotations in the second stage of the translation method (see Section 1.2.2) are annotated with pre_set, post_set and exc_set constructs. These set notations represent the events that happen when a program enters or exits a method or throws an exception. These events initiate a transition in the property automaton and therefore are part of the JML annotation describing the JML security policy. An in depth explanation is give in Section 6.3.

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

Figure 2.3: Syntax of Java method, class and program

Last, a program consists of a set of classes. A *Class* has a name and a super class from which this class is derived. Furthermore, a *Class* contains static or non-static fields/ghost variables. Last, it contains a set of methods and invariants, which are predicates that hold true throughout the lifetime of the class.

2.4 Program state

A program state is a summary of the program's data structure. The data structure of a program is a collection of (static) fields and local variables. A field, shown Figure 2.3, is a variable at class level and can be either static or non static. Furthermore, a local variable can either be a method parameter or a variable declared inside a method.

$$
\begin{array}{r}
\mathsf{pstate} := [\mathsf{exc} : \mathit{Excp}_i, \mathsf{store} : \mathit{PStore}] \\
\mathit{PStore} := [\mathsf{classes} : \mathit{Class} \rightarrow \mathsf{PClassStore}, \mathsf{objects} : \mathsf{Ref} \rightarrow \mathsf{PObjectStore}, \\
\mathit{loc\_vars} : \mathcal{N} \rightarrow \mathcal{V}]] \\
\mathsf{PClassStore} := [\mathsf{classname} : \mathcal{N}, \mathsf{statfields} : \mathcal{N} \rightarrow \mathcal{V}] \\
\mathsf{PObjectStore} := [\mathsf{instfields} : \mathcal{N} \rightarrow \mathcal{V}]
\end{array}
$$

Figure 2.4: State definitions

The pstate represents the current program state. When an exception is generated, it is stored in the program state.
The *PStore* is the collection of all static and non static fields of a certain class and also maintains a single local variable stack. The PClassStore contains all the static fields of a class, and occurs once per class.
The PObjectStore can be dynamically created upon constructing a new object of a certain class and the amount of PObjectStore corresponds to the amount of created objects.

This concludes the syntax and program state definition of the simplified Java programming language. The introduction of objects mainly affects the expression and method definition, because the assignment and the lookup of a variable can refer to an object variable. Furthermore, it introduced the concept of static and non-static variables, but it did not include static methods and classes.

Chapter 3

Security policies and security automata

This chapter discusses the formal definition of a security policy, the definition of a mechanism enforcing it and the definition of a security automaton, which is an instance of an enforcement mechanism. The enforcement mechanism is equivalent to the Execution Monitor, described in Section 1.1. Furthermore, the derivation of a security automaton from the security policy will illustrate that not all security policies are representable as a security automaton.

3.1 Security policy

A security policy, in relation to a program, describes an execution, which is unacceptable for one reason or another [11]. For example, a CPU scheduling algorithm in an operating system should not deny a program access to the CPU indefinitely. Or certain resources can only be accessed when an authenticated user has enough privileges.
Furthermore, the practicality of a security policy depends on its enforceability such that its next step can only be determined based upon the current program state and the generated event. Therefore, the translation method of Chapter 6 is only useful for security policies which are enforceable .
The adherence of an enforceable security policy can be verified by an Execution Monitor (EM). The EM monitors a piece of software, called the program, which terminates the execution when the program is about to violate the security policy.

Formally, a security policy is specified by giving a predicate on sets of executions. A program S satisfies a security policy $\wp$ if and only if $\wp(\sum s)$ equals true, where $\sum_S$ is the set of all possible runs of program S. In other words, a security policy is a function which evaluates a given run by program S to be true or false. When false, the security policy is violated.

As indicated earlier, the security policy $\wp$ can for example be implemented using an enforcement mechanism in an EM. The EM is defined as

$$\wp(\pi) : (\forall \sigma \in \pi : \hat{\wp}(\sigma)) \tag{3.1}$$

i where π is a subset of all possible runs $\wp(\sum s)$

ii σ is a run of finite or infinite length and

iii $\hat{\wp}$ formalizes the criteria used by the enforcement mechanism for deciding whether or not to terminate an execution that otherwise would violate the security policy being enforced.
In other words, the property $\hat{\wp}$, representing a security policy $\wp$, evaluates true for all runs π of a program when the program does not violate the security policy.

3.2 Security property

Property $\hat{\wp}$ is a security property if: [11]

- it evaluates each run σ of program π individually independent of other runs;
- is prefixed closed;
- rejects an unacceptable run within a finite period of time.

A set of runs is considered to be prefixed closed, when an acceptable run σ does not contain a prefix σ' which is rejected by the same property $\hat{\wp}$. If not, this would imply that a violation of the security policy would be ignored, since it is assumed that the program halts when a violation is detected. This leads to the conclusion that an enforceable security policy has to pass the criteria of a security property. However, it does not always hold that a security property is enforceable, due to practical reasons i.e. the evaluation of a run σ by $\hat{\wp}$ is too complex. For example a security policy, such as a web server should eventually reply to a pagerequest is not enforceable. The monitor of the web server should wait indefinitely before it can detect a violation.

3.3 Security automata

The enforcement mechanism in an Execution Monitor, described in the previous section, halts a program when a the security property $\hat{\wp}$ evaluates to false. A recognizer for a security property $\hat{\wp}$ can be represented in the form of a security automaton. Security automata are a class of Büchi automata that accept safety properties and are similar to ordinary non-deterministic finite state automata [11].

Formally, a security automaton is defined as: [16]

- a countable set Q of automaton states;

- a countable set $Q_0 \subseteq Q$ of initial automaton states;

- a countable set I of input symbols, and;

- a transition function, $\delta : (Q \times I) \rightarrow 2^Q$.

The set of input symbols is defined by the security policy and may for example correspond to system states, atomic actions or state/action pairs. A run is defined to be a sequence of input symbols $S_1S_2S_3\ldots$, where the security automaton starts in an initial state Q_0.

The transitions from state to state occurs in the following way: 1) The Security Automaton is currently in state Q.
2) The SA reads the next input I 3) Based on the input I and current state Q, it evaluates the possible transitions. This is formally defined as $\bigcup_{q \in Q'} \delta(q, s_i)$, where Q' is the set of possible next states and $\delta(q, s)$ is the transition function. Thus for all states of q the transition $\delta(q, s)$ evaluates to true are part of Q'.
4) If multiple states are possible, one state is chosen at random or when Q' contains only 1 state, it indicates the next state.
Furthermore, note that the security automaton can accept runs of finite and as well as infinite lengths.

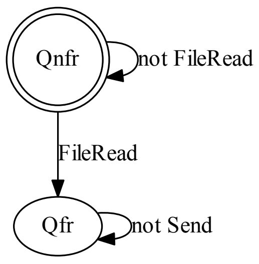

Figure 3.1: No send after fileread

A security automaton can be represented graphically, as shown in Figure 3.1 and in text form, as shown in Figure 3.2. This example represents the security policy that a file cannot be send after it is read by a user. It starts out in state Qnfr representing that the file has not been read. From here the file can either be read or not read. When the file is not read, the state remains the same and when the file is read it transits to a final state. In this state, the user is not allowed to send the file anymore. Thus the security automaton only describes the desired behavior.

```
**state vars** state : {0,1} **initial** 0

**transitions** **not** FileRead ∧ state = 0 → **skip**
        FileRead ∧ state = 0 → state := 1
        **not** Send ∧ state = 1 → **skip**
```

Figure 3.2: Access control automaton text example

This concludes the description of a security automaton as described by Schneider [11, 1].

Chapter 4

Property automata

This chapter gives the formal definition of a Property Automaton (PA). The original PA of Huisman and Tamalet is adapted from the definition of the Security Automata (SA) described in Chapter 3. [7] As with the SA, the PA can be used to express security policies, such as a file cannot be send after it is read. The PA formalization of Huisman and Tamalet is limited to a static methods. This chapter adapts the formalization to support multiple instance methods distinguishing between different receivers.

First, an informal description of a PA will be given, followed by an example of a PA, which is given in textual and graphical form. This example will also be used to illustrate the translation method of Chapter 6. Next, the formal definition of a PA is given. Followed by more examples of PAs, which demonstrate the abilities and limitations of the adapted PA definition. Last, the structure of the PA state and the semantics of PA state transitions will be discussed.

4.1 Informal description

The PA can be thought of as a Finite State Machine (FSM). The PA consists of a set of control points which are connected via transitions.

The following terms are related to a PA:

control point A Control Point(CP) is a state of the PA. The set of starting CPs are called initial states.

PA variable A PA variable has a name and a type and is associated with a PA. (i.e. a counter) The type set is limited to the one defined in section 2.1. Furthermore, a PA can be set in an action and can be retrieved in a guard or an action.

class variable A class variable is a PA variable with a name and a classtype. The class variable is associated with a single object instance at run-time and limits the execution of transition with the same Classvar to the same object instance.

transition The transitions describe the change in behavior of the security policy. It has a source CP and a destination CP. A transition is initiated by an event and limited by a guard and class variable. The corresponding actions update the PA variables, when taking the transition.

event An event represents the entry, normal exit or exceptional exit of a method in one of the objects, which are being monitored. When an event initiates a transition, the object causing the event should match the class variable of that transition.

guard A guard gives the condition under which a transition can occur and can depend on:

i) the automaton state
ii) the program variables.
iii) the argument of the method

action An action indicates how the automaton state is updated. A special reset() action is defined, which sets all class variables to null.

As with a FSM, a PA starts in one of its initial states and executes one transition after another. The only difference with a FSM, is that the transitions are initiated by program events (i.e. as the entry or exit of a method). Once an event is evaluated, it select the transitions matching the event, for which its guard evaluates true and the class variable, if set, matches the object generating the event. This process repeats itself for each event generated by the program. If all events have a matching transition, the program is not violating the security policy.

4.2 Example 1: max 3 access

In this example, there is a set of terminals t, which are used to verify a customers pincode corresponding to a bank card. For simplicity it is assumed that there is only one card and one pincode. Furthermore, the entered pincode can either be entered correct or incorrect. The security policy states that after 3 wrong tries at possibly different terminals, the bank card is blocked.
Figure 4.1 gives a graphical and Figure 4.2 gives a textual overview of the PA for example 1.

This automaton only describes the allowed behavior and comprises of 3 states, the initial state s0 and the regular states s1 and s2. The arrows in Figure 4.1 are transitions. A transition has the following format:

$$\mathsf{eventType(classVar.method)?guard} \rightarrow (\mathsf{action}_1; \mathsf{action}_2 \ldots) \tag{4.1}$$

The PA monitors certain events and depending on its current state and the event a set of transitions are available. A transition is further selected and executed depending on its guard and its class variable. A guard is an expression, which resolves to either true or false. Furthermore, a class variable is a reference to the object which generated the event. In case of figure 4.1, the pay event is generated by a terminal after it checked if the customer pincode was correct or incorrect.

From the set of available transitions, only one transition is chosen for which the guard holds and the class variable corresponds. However, if a class variable of a transition is not set, the class variable is ignored as guard. This results in the possibility of transitions sharing the same event, but having a non overlapping guard.

Once a transition is selected, its class variable is set equal to the object generating the event. This forces a transition containing the same class variable to be generated by the same object. See figure 4.4 for an example where class variables are reused. In case of figure 4.5, the class variables $t1, t2$ and $t3$ refer to the terminals at which a customer can enter his pin code.. The class variables are only set when an incorrect pincode is entered.

When a transition is taken, its corresponding actions are executed. One special action is the reset() action, which sets all the class variables to null. In figure 4.1, the reset() action is defined for the transitions where the pincode has been entered correctly. The reset() action clears all the class variables of type Terminal, such that new transactions can be performed at any terminal.
If this reset() action would have been left out, the PA of figure 4.1 would force the customer to enter his pincode at terminal $t1$ during the next transaction.

4.3 Property automaton definition

This section introduces the formal definition of the Propery Automaton (PA), shown in figure 4.3.
A *PA* has a name and monitors a set of classes clnames. Furthermore, a *PA* consists of a set of cps of which a subset is identified as initial cps in which the monitor starts.
The set of transitions *Trans* allows the *PA* to transit from one *CP* to another.
A class variable classVar has a name and a ClassType and is associated with an object instance of ClassType at run time.
The set of event on its turn is limited by the monitored methods of the set of classes.
Last, the *PA* contains automaton (pa_var_decl) and program (prog_var_decl) variables which are PA and program variables. These variables can be used in the guard.

The transition relation formalizes the allowed behavior from one *CP* to another *CP*. Furthermore, a transition *Trans* is characterized by an event generated by a method entry, normal or exceptional exit during runtime. The universe of possible transitions is limited by the set of events and class variables. The associated guard specifies under which conditions an event is allowed to occur. When a transition is taken, a set of actions can be given to update the PA variables. The 'target' attribute in the 'action' definition refers to a ghost variable stored in the pa_var_decl attribute of the PA definition and is set to the associated expression. Last, the reset() method is defined as a special action. When called, it erases all the class variables of the PA.

An event consists of a class variable ClassVar, an event type etype and a method name mname. The class variable indicates which *Class* can generate this event, the method name indicates which method of that *Class* can generate the event and the event type indicates if the event is generated at the entry, exit of the method or if the methods generates an exception. As indicated earlier, an event thereby characterizes the transition between two PA states.

4.4 More examples

The following section discusses 2 additional examples to demonstrate the abilities and limitations of the new PA definition. Figure 4.4 describes a buffer on which a put and get operation can be performed. Figure 4.5 represents a transaction where a terminal is credited and a cash card is debited. Note, that these two examples are not given in a textual representation.

4.4.1 Example 2: buffer get and set

The following example represents a buffer on which a get and a set operation can be performed. Since the Java programming language supports object instantiation, multiple instances of the buffer object can be active while monitoring the program. As explained earlier, only one PA is active to monitor multiple buffer objects. Due to this limitation only one buffer at a time can perform a put/get operation. Thus the put/get operations of each Buffer instance cannot be interleaved in this notation.

It is possible to define a single PA which contains all possible interleaved runs for a fixed amount of buffers, where each buffer is modeled as a class variable. However, the size of the PA increases drastically with the addition of a Buffer instance. Therefore this PA definition is not very scalable for situations where interleaving of class methods among a large set of object instances is required. A possible solution to this problem is to parameterize the PA, such that objects with an identical behavior can be monitored by a duplicate of the original PA. In case of this example, it would mean that every buffer would get a duplicate PA identified by a parameter.

4.4.2 Example 3: A terminal and a cashcard

The third example describes a transaction between the Terminal and a Cash card and is shown in figure 4.5. In this case, the cash card is debited and the terminal is credited. Note that upon returning to the initial state, the class variables are reset, such that the next transaction is not limited to same terminal and cashcard.

This example shows that the PA definition supports monitoring of methods of different classes.

4.5 Property automaton state definitions

The following section introduces the Property Automata (PA), annotated and monitored state definitions. The PAState represents the current state of the PA. How the data structure is manipulated by transitions of the PA is shown in Section 4.5.1 Furthermore, the annotated and monitored state definitions are defined. These state definitions are used for the second and third stages of the translation method of Chapter 6. But first the definition of the property automaton transition is given in the next section.

4.5.1 Property automaton state and transition function

The state space of a PA consists of its current CP and an image of the automaton variables called the store.
$PAState = [\mathsf{current} : CP, \mathsf{store}_{\mathsf{A}} : \mathsf{store}]$

The transition function Δ_a, which updates the PAState depends on its current PA state, the program state, the event (method entry, norm_exit, excp_exit) occuring and the value returned by the method. As long as the transition function Δ_a does follow the PA behavior, the security policy is not violated.
The transition function first selects a transition t, which i) starts in the current state; ii) shares the same event which has been generated; iii) for which it guards holds true. If multiple transitions match these qualifications, a transition is chosen arbitrarily.
After the transition has been selected, the transition function i) updates the current control point to the destination control point of the transition; ii) updates the PA variables based upon the action, the event and the current PA and program variables;

The PA definition of figure 4.3 was altered by the introduction of class variables and the special reset() action.
The class variables of a PA are stored as PA variables. Furthermore, the class variables in the event definition of a PA are symbolic links to their corresponding PA variable. Thus when in transition function event e is compared to an event of a transition t, it automatically is compared to its corresponding PA variable.
After a transition has been executed, the class variables in form of PA variables are updated by the apply function using the event e.
Last, the reset() action clears all of the class variables in the form of PA variable. For simplicity, the clearing of the class variables is performed by the apply function.

4.5.2 The annotated and the monitored state

The AState represents the annotated state in which the Java program is annotated with the set notations.
The annotated state can also contain ghost variables, which are represented as regular static and non-static fields in the PClassStore and the PObjStore of the program state. The advantage is that in the semantics of variable lookup

in Section 5.2, no distinction has to be made between ghost and regular fields. Furthermore, it is assumed that the well-formedness of the ghost variables are checked at compile time, thereby allowing this generalization at run time.

The monitored program state is associated with a PA. It keeps track of the PAState as before, introduces the program state, the ghost variables and a stuck flag. If the program raises a not specified event, the stuck flag will be set permanently. This then indicates that the program violated the behavior of the PA.

4.6 Security automata vs property automata

The Security Automata definition in the previous chapter is similar to the Property Automaton definition of the previous section.
The SA and PA match on the following points:

Control Points - both automata have a set of control points, which represents the state space of the automaton;

Initial Control Points - both automata have a set of initial control point in which the automaton can start;

Transitions - both automata have transitions which changes the state of the automaton. However a transition from the same stat in a SA is only distinguishable by a single action, while the PA can distinguish transitions based upon events, guards and class variables.

Automaton state The automaton state of a SA only depends on its current control point, while the automaton state of a PA can also depend on PA variables.

Thus in relation to the SA, the PA can be thought of as a SA where the input definition I is replaced by an event, a guard and a class variable and thereby making the transition function more complex.

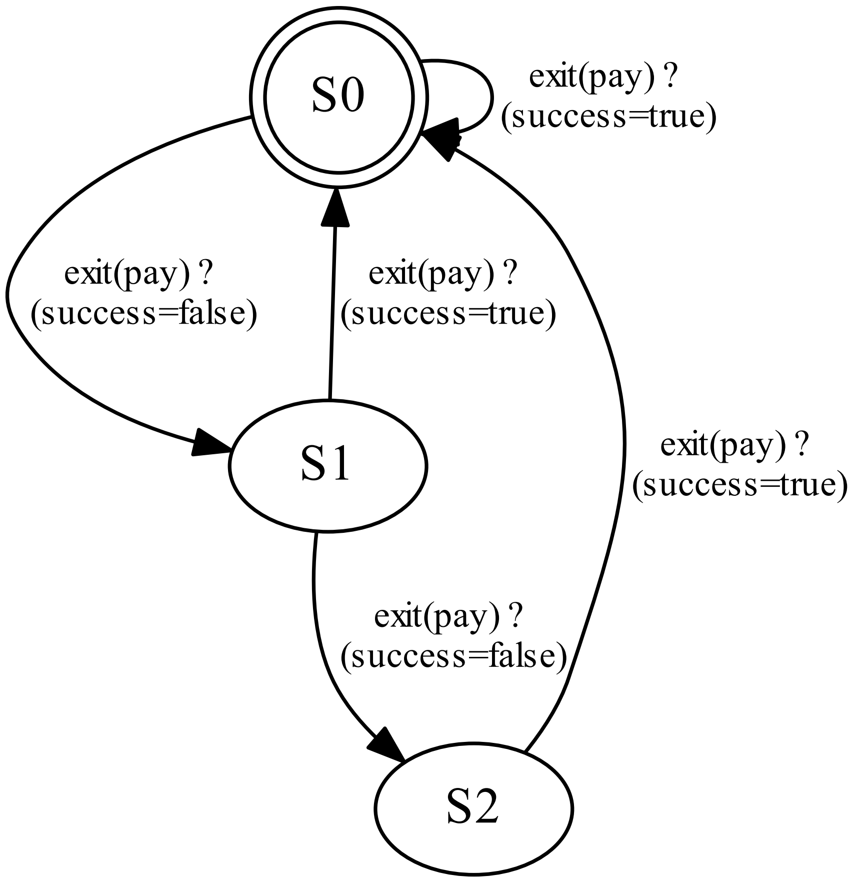

Figure 4.1: Example 1: max 3 tries; Terminal t1,t2,t3

```
PA = (#
        name := Max3tries, clnames := {terminal},
        classVars := {varName = t1, t2, t3 : ClassType = Terminal},
        cps := {s_0, s_1, s_2}, init := s_0 #),
      events_PA :=  {(# etype := exit, classType := {Terminal}, classVar  in{t1, t2, t3},
                     mname := pay #),
      pa_var_decl :=  ∅, sprog_var_decl := ∅,
      trans := {   (#  source := s_0, dest := s_1, guard := success = false,
                       event := (# etype := exit, classVar := t1, classType = Terminal,
                       mname := pay #) #),
                   (#  source := s_1, dest := s_2, guard := success = false,
                       event := (# etype := exit, classVar := t2, classType = Terminal,
                       mname := pay #) #),
                   (#  source := s_0, dest := s_0, guard := success = true, action := reset(),
                       event := (# etype := exit, classVar := t1, classType = Terminal,
                       mname := pay #) #),
                   (#  source := s_1, dest := s_0, guard := success = true, action := reset(),
                       event := (# etype := exit, classVar := t2, classType = Terminal,
                       mname := pay #) #),
                   (#  source := s_2, dest := s_0, guard := success = true, action := reset()
                       event := (# etype := exit, classVar := t3, classType = Terminal,
                       mname := pay #)
      }
#)
```

Figure 4.2: Example 1: Max 3 tries

```
       PA  = [#  name : 𝒩, clnames : 𝒫(𝒩), cps : 𝒫(CP),
                 init : CP, trans : 𝒫(Trans), classVars : 𝒫(classVar),
                 events : 𝒫(Event), prog_var_decl : 𝒫(Decl), pa_var_decl : 𝒫(Decl) #],
     Trans = [#  source, dest : CP, event : Event, action : 𝒫(Action),
                 guard : PAState × PState × (𝒱 | ℰ) → 𝔹 #]
    Action = [#  target : 𝒩, expr : Expr #] | reset(),
     Event = [#  classVar : ClassVar, etype : (entry | exit | exc_exit), mname : 𝒩 #]
  ClassVar = [#  varName : 𝒩, classType : Class #]
```

Figure 4.3: Property automaton formalization

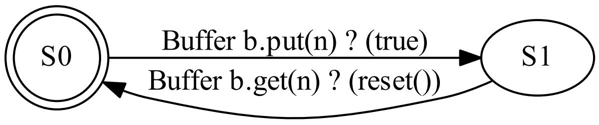

Figure 4.4: Buffer get and set: Buffer b

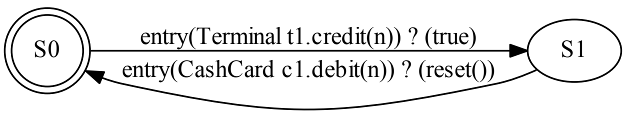
S0
S1
entry(Terminal t1.credit(n)) ? (true)
entry(CashCard c1.debit(n)) ? (reset())

Figure 4.5: A terminal and a cashcard

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

Figure 4.6: PA transition function

$$
\begin{aligned}
AState &= [\mathsf{pstate} : PState] \\
MState &= [\mathsf{pa\_state} : PAState, \mathsf{pstate} : PState, \mathsf{stuck} : \mathbb{B}]
\end{aligned}
$$

Figure 4.7: State definitions

Chapter 5

Operational semantics

This chapter introduces the relevant operational semantics of the non-annotated, annotated and the monitored program using the simple Java syntax of Chapter 2. The formalization of the Java operational semantics in this chapter are to a great extend adapted from Obheim and also from Huisman and Tamalet [7, 13]. First, the way methods are evaluated by Java will be described. Second, the lookup of variables is described. Third, the way objects are created and initialized is described. Fourth, the assignment of values to variables is discussed. Lastly, the semantics of the monitor is discussed.

5.1 Method evaluation

The method evaluation is invoked by a method call on a method with a method name mn and an object reference o. Furthermore, a method parameter p is passed and resolved as a local variable act. After that the method definition is retrieved and stored in the variable md.
Before the method body is executed, the old local variables are backed up, such that they can be restored at the end of the method evaluation. The local variable store is replaced by the method parameter, the local method variables and the result of the method.
Next, the method body is evaluated and the result value is returned. Finally just before exiting the method evaluation the postconditions are checked and the old local variables are restored.

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

Figure 5.1: Method evaluation semantics

Despite the introduction of objects in the Java language of Chapter 2, the operational semantics of method evaluation has not been changed from the specification of Huisman and Tamalet [7]. However the method evaluation semantics describes the behavior of object methods instead of static methods.

5.2 Variable evaluation

The operational semantics of variable evaluation is identified by the statement Var(target, $\mathcal{N}$). When the target is set to null or not given, it is assumed that either the variable $\mathcal{N}$ is a PA variable or a local method variable of the method being executed.

$$\frac{\begin{array}{c}\sigma_0.\mathsf{prog\_state.exc} = \bot \\ v_1 = pa\_state.store_a(\mathcal{N}) \\ v_2 = pstate.pstore.local\_vars(\mathcal{N}) \\ v = (v_1 \neq \bot)?v_1 : v_2 \\ P \vdash \langle \mathcal{N}, \sigma_0\rangle \triangleright \langle v, \sigma_0\rangle\rangle\end{array}}{P \vdash \langle \mathsf{Var}_{\mathbb{I}|\mathbb{B}|\mathbb{R}}(target(= null), \mathcal{N}), \sigma_0\rangle \triangleright \langle v, \sigma_0\rangle\rangle}$$

Figure 5.2: PA and local method variable evaluation

If the target variable is set, it is assumed that it refers to an object, where $\mathcal{N}$ is a static or not static field variable of that object. This field variable can refer to a Java field or the annotated JML ghost field variable. As described in Section 4.5.2, this generalization simplifies the handling of JML ghost fields variables.

The first step in the evaluation is to determine if the target object in the call statement is defined or not.
If not, the evaluation function will first retrieve the corresponding PA ghost variable $\mathcal{N}$ and thereafter retrieve the value of the local method variable name. It is assumed that only one of the two variables is defined and returned. The PA ghost variables are prioritized over the local variables.

$$\frac{\begin{array}{c}\sigma_0.\mathsf{prog\_state.exc} = \bot \\ objStr = \sigma_0.Pstore.object(target) \quad classStr = \sigma_0.Pstore.classes(target) \\ v_1 = objStr.instfields(\mathcal{N}) \quad v_2 = classStr.statFields(\mathcal{N}) \\ v = (v_1 \neq \bot)?v_1 : v_2 \\ P \vdash \langle \mathcal{N}, \sigma_0\rangle \triangleright \langle v, \sigma_0\rangle\rangle\end{array}}{P \vdash \langle \mathsf{Var}_{\mathbb{I}|\mathbb{B}|\mathbb{R}}(target(\neq null), \mathcal{N}), \sigma_0\rangle \triangleright \langle v, \sigma_0\rangle\rangle}$$

Figure 5.3: Class static, non-static field variable evaluation

If the target is defined, both object and the class store of the target object are retrieved. Next, both the class and object store are evaluated for $\mathcal{N}$, where it is assumed that only one of both is defined. Last the evaluated value is returned, where the non static fields are prioritized.

5.3 Object creation

This formalization describes the global steps taken, when a new object is created.
First the static class store is initialized by calling the init function as described in the next section. Notice that the static class store is only initialized when it is not already created.
Next, the createPObject function is called, which creates the object store dynamically and calls the class constructor to initialize the object store. After the execution of createPObject has finished it evaluates into a reference r to the object and it extend the program state with the PObjStore.
Last, the object stores in the Java programming language are automatically destroyed by the garbage collector. However, the simplified Java language does not formalize the garbage collector and the destruction of the objects.

$$\frac{\begin{array}{c} P \vdash \langle init(cname), \sigma_0\rangle \triangleright \langle v, \sigma_1\rangle\rangle \\ P \vdash \langle createPObject(cname, eval(p)), \sigma_1\rangle \triangleright \langle r, \sigma_2(PStore.Objects \cup \{r \rightarrow PObjStore\})\rangle \end{array}}{P \vdash \langle new(cname : N, p : Expr), \sigma_0\rangle \triangleright \langle r, \sigma_2)\rangle}$$

Figure 5.4: Object creation

5.4 Static class initialization

$$\frac{P \vdash \langle init(cname), \sigma_0\rangle \triangleright \langle v, \sigma_1(PStore.classes \cup \{cname \rightarrow PClassStore\}))\rangle}{P \vdash \langle init(cname : N), \sigma_0\rangle \triangleright \langle v, \sigma_1)\rangle}$$

Figure 5.5: Semantics for initializing PClasstore

As indicated in the previous section, the init function initializes the static class store of Class cname. It only does so if the static class store has not already been initialized, which is checked in the beginning by the init function.
Furthermore, the class stores in the Java programming language are automatically destroyed by the garbage collector. But since this formalization does not contain a garbage collector, the destruction of the static class stores is not specified.

$$\frac{\begin{array}{c} rdy = initd(cname) \\ P \vdash \langle init(cname), \sigma_0 \rangle \triangleright \langle v, \sigma_1 \rangle \rangle | \sigma_0 = \sigma_1 \end{array}}{P \vdash \langle init(cname : N), \sigma_0 \rangle \triangleright \langle v, \sigma_1 \rangle \rangle}$$

Figure 5.6: Semantics for an already initialized PClasstore

$$\frac{\begin{array}{c} P \vdash \langle e, \sigma_1 \rangle \triangleright \langle v, \tau_1 \rangle \qquad P \vdash \langle getRef(t,n), \tau_1 \rangle \triangleright \langle r, \tau_2 \rangle \\ \text{if } \tau_2.\mathsf{pstate.exc} = \bot \text{ then } \sigma_2 = \tau_2(copy(r,v)) \text{ else } \sigma_2 = \tau_2 \end{array}}{P \vdash \langle Assign(t,n,e), \sigma_1 \rangle \triangleright \langle v, \sigma_2 \rangle}$$

Figure 5.7: Assign evaluation semantics

5.5 Assignment evaluation semantics

The assignment evaluation semantics formalizes the copying of a expression result in to a variable. First, the expression e is evaluated into a resulting value v. Because, the assignment operation is also an expression, the result of the assignment operation will be v.
Second, the getRef method retrieves for a target class t and variable name the reference to the variable. Depending on whether the target class is set or not, the getRef method will evaluate the reference in a similar fashion as it retrieves the value as described in Section 5.2. After the resulting reference is returned, the copy function copies value v into the address of reference r. Last, this formalization does not specify the event in which a reference of the indicated variable cannot be retrieved.

5.6 Monitored program semantics

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

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

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

where

$$\begin{array}{lcll} \gamma_{\text{PA}}(ev)(P, md, act, \sigma) & = & \text{let} & e = (\#\, \mathsf{etype} := ev, \mathsf{mname} := md.\mathsf{name}\, \#), \\ & & & \tau = \Delta_{P.\mathsf{pa}}(\sigma.\mathsf{pa\_state}, \sigma.\mathsf{prog\_state}, e, act) \text{ in} \\ & & \multicolumn{2}{l}{\text{if } \sigma.\mathsf{stuck} \lor \tau = \bot \text{ then } \sigma(\mathsf{stuck} := \mathsf{true}) \text{ else } \sigma(\mathsf{pa\_state} := \tau)} \end{array}$$

Figure 5.8: Monitored Program Semantics

The monitored program semantics gives the formalization of handling the pre set annotation. The introduction of objects in the simplified Java language and PA definition is dealt with by the monitor by matching the class variables of the PA transitions with the objects generating an event. The matching of the class variable with the event generating object is performed in the translation method of 6.3 by adding additional guards. Since this formalization has a generic method of handling guards, it does not affect the monitored program semantics.

Chapter 6

Translation method

This chapter covers the translation method described by Huisman and Tamalet and extends it to support objects [7]. Unlike Huisman and Tamalet, this assignment does not mathematically proof the equivalence of the old and new program. The translation method transforms a security policy, expressed as a Property Automaton (PA), into JML annotations of the Java program. Next, the annotated Java program can either statically or dynamically verify the adherence of the program to the security policy, using the Mobius Verification Environment [2].

The dynamic verification method compiles the annotated Java program into Java byte code and checks the adherence to the security policy at run time. The static verification method can use the ESC2/Java tool to check the adherence by analyzing the source code and not executing or compiling the program. However, the ESC2/Java tool can handle only programs where the behavior of the program is fully annotated in JML.This translation method generates only part of the JML annotations and therefore should be extended in order to be supported by ESC/Java2. Nonetheless, assuming that the field of static verification using JML evolves, the static verification has the potential to automate the verification of security policies.

The translation method is split up into 3 stages:

stage 1 completion of the Property Automaton;

stage 2 annotates the monitored class methods with set-annotations;

stage 3 in-lines the set-annotations of each method into the method body.

The next sections explain the stages in more detail.

6.1 Stage 1: automaton completion

The first step in the translation method is the the automaton completion. This step adds a halted state to the PA and directs events which violate the security policy to the halted state. Thereby, the completed automaton defines the behavior which adheres and which violates the security policy. The original automaton completion of Huisman and Tamalet remains intact [7].

The formal definition of automaton completion is as followed:

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

P is defined to be the monitored program consisting of a program $P.\mathsf{program}$ and a PA $P.\mathsf{pa}$. The program remains intact, while the $P.\mathsf{pa}$ definition is transformed to a complete automaton and is updated in the monitored program definition.

The completion function first extends the old PA by adding a halted state and adding new transitions towards the halted state. All of these new transitions have in common that they represent the disallowed behavior of the PA and can be split up into two types. The completion function has the following definition:

Stuck CP A new CP is added to cps, the set of control points of PA (Section 4.1)
$p.\mathsf{cps} \cup \{\mathsf{halted} : CP\}$

Transition type 1 create a new transition starting in a CP and with the halted CP as destination, for which an event-type, with the same event type and method name, has not yet been defined. In the event-type the class variable is ignored, such that it does not duplicate a transition for each class variable.

$$\begin{aligned} &p.\mathsf{trans} \cup \{t = [\#t.\mathsf{source} = cp_{\mathsf{source}}, t.\mathsf{dest} = \mathsf{halted}, t.\mathsf{event} = e,\\ &\qquad \mathsf{tr.guard} = (\mathsf{true} \to \mathsf{Skip})\#] \mid cp_{\mathsf{source}} \in p.\mathsf{cps} \land\\ &\qquad\quad e \in p.\mathsf{events} \land \neg\exists\{\mathsf{tr2} \mid \mathsf{tr2} = cp_{\mathsf{source}} \land \mathsf{tr2}.e.\mathsf{etype} = e.\mathsf{etype} \land\\ &\qquad\qquad \mathsf{tr2}.e.\mathsf{mname} = e.\mathsf{mname}\}\}\end{aligned}$$

Transition type 2 for each set of transitions starting in the same CP and sharing the same event. Add a new transition from the CP to the halted state, with the same event and a guard which is the negated conjunction of the guards of the transition set.

$$\begin{aligned} &p.\mathsf{trans} \cup \{t = [\#t.\mathsf{source} = cp_{\mathsf{source}}, t.\mathsf{dest} = \mathsf{halted}, t.\mathsf{event} = e,\\ &\qquad \mathsf{tr.guard} = \neg \bigwedge\{\mathsf{tr.guard} | \mathsf{tr} \in \mathsf{trset}\}\#] \mid\\ &\qquad \mathsf{trset} = \{\mathsf{tr2} \mid \mathsf{tr2} \in p.\mathsf{trans} \land \mathsf{tr2}.cp = cp_{\mathsf{source}} \land \mathsf{tr2}.e = e\} \land\\ &\qquad \mathsf{trset} \neq \emptyset \land cp_{\mathsf{source}} \in p.\mathsf{cps} \land e \in p.\mathsf{events}\}\end{aligned}$$

The next section will illustrate the definition via an example.

6.2 Example of completed automaton

A graphical version of the completed PA is shown in Figure 6.1 and the textual version is shown in Figure 6.2. The completed PA consists of the allowed and disallowed behavior of the Max3Tries example and the notation is similar as in Figure 4.1. The allowed behavior is the case that a customer enters the correct code on a terminal within the limit of 3 tries. The disallowed behavior represents a violation of the security policy.

As show in Figure 6.1, the first step adds halted state. Next, two types of transitions are added to the Property Automaton and are directed to the halted state.
The first type of transitions are all the transitions where the guard is set to true and an event-type, which does not appear in any other transition from the same state. Note that the event-types only differ in the class variable defined in the event.
The second type of transition occurs when the customer performed a wrong pay for a third time and goes from s2 to the halted state.

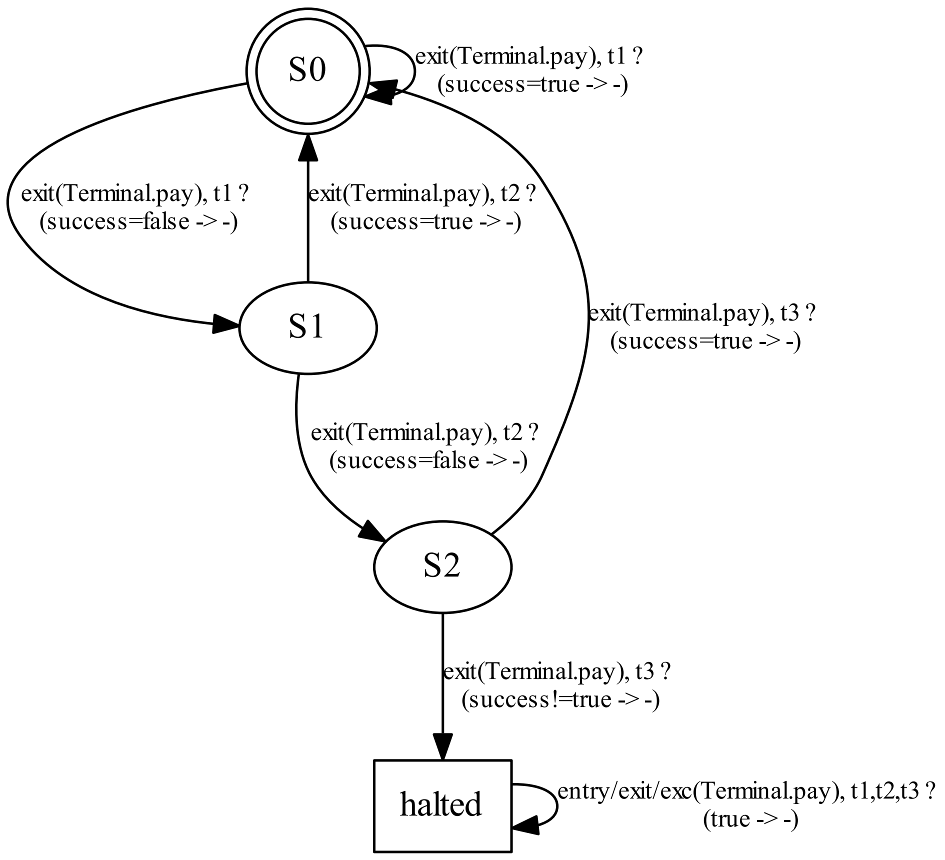
S0
exit(Terminal.pay), t1 ? (success=true -> -)
exit(Terminal.pay), t1 ? (success=false -> -)
exit(Terminal.pay), t2 ? (success=true -> -)
S1
exit(Terminal.pay), t3 ? (success=true -> -)
exit(Terminal.pay), t2 ? (success=false -> -)
S2
exit(Terminal.pay), t3 ? (success!=true -> -)
halted
entry/exit/exc(Terminal.pay), t1,t2,t3 ? (true -> -)

Figure 6.1: Completed Property Automaton

$PA =$

(# name := Max3tries,
clnames := $\{$Terminal$\}$,
cps:= $\{s_0, s_1, s_2,$ halted$\}$,
init:= s_0
classVars:= $\{$ (# t1 : Terminal #), (# t2 : Terminal #), (# t3 : Terminal #)$\}$
event:= $\{$(# classVar: (# t1 : Terminal #), etype: exit, mname: pay #),
(# classVar: (# t2 : Terminal #), etype: exit, mname: pay #),
(# classVar: (# t3 : Terminal #), etype: exit, mname: pay #)$\}$
pa_var_decl := $\emptyset$,
prog_var_decl:= $\emptyset$,
trans := $\{$ (# source := s_0, dest := s_1, guard := $success = false$,
event := (# classVar : (# $t1$: $Terminal$ #), etype : exit, mname : pay #) #),
(# source := s_0, dest := s_0, guard := $success = true$,
event := (# classVar : (# $t1$: $Terminal$ #), etype : exit, mname : pay #) #),
(# source := s_1, dest := s_2, guard := $success = false$,
event := (# classVar : (# $t2$: $Terminal$ #), etype : exit, mname : pay #) #),
(# source := s_1, dest := s_0, guard := $success = true$,
event := (# classVar : (# $t2$: $Terminal$ #), etype : exit, mname : pay #) #),
(# source := s_2, dest := s_0, guard := $success = true$,
(# classVar : (# $t3$: $Terminal$ #), etype : exit, mname : pay #) #),
(# source := s_1, dest := $halted$, guard := $success = false$,
event := (# classVar : (# $t3$: $Terminal$ #), etype : exit, mname : pay #) #),
(# source := $halted$, dest := $halted$, guard := $true$,
event := (# classVar : (# $t1, t2, t3$: $Terminal$ #), etype : exit, mname : pay #) #)
$\}$
#)

Figure 6.2: Completed automaton example 1: Max 3 tries.

6.3 Stage 2: generating set annotations

The second stage involves generating the method level set-annotation for each class and method of the Java source code. Together these annotations represent the PA in annotated form. The definition of the second step in the translation method is shown in Figure 6.3 and adapted from Huisman and Tamalet [7].

The annotation generation process generates the annotation in the following way:

$\alpha_2(p)$ For each class c and Property Automaton a of program p, start annotating the class by calling $\alpha_{2,\mathcal{C}}(c,a)$.

$\alpha_{2,\mathcal{C}}(c,a)$ If class c is not monitored by PA a, leave the source code intact. Otherwise for class c, add the PA variables to the ghost variables of the class. A description of PA variables is given in Section 4.1. The ghost_vars of Class c are JML ghost variables, which are used by JML to data which is not part of the regular program. An example of such a variable is the current control point CP, which is only part of the annotated state.
Furthermore, a JML invariant is added, which makes sure that the current cp is not equal to the halted state. In other words the annotated PA does violate the security policy.
Next, the annotation process $\alpha_{2,\mathcal{M}}(c,m,a)$ for each method m of class c is started and $\alpha_{2,\mathcal{R}}(a)$ adds a reset method.

$\alpha_{2,\mathcal{R}}(a)$ This rule generates a reset method, which sets all class variables equal to Null. The reset method is a special action, which can be called when taking a transition. The class variables themselves are references to objects and associated with transitions. The first time the transition is taken with a particular class variable, the class variable is set equal to the object reference of the event generating object. When another transition is taken with the same class variable, the object initiating the transition should be the same as the one stored in the class variable. This is valid for all upcoming transitions, until this reset method is called.

$\alpha_{2,\mathcal{M}}(c,m,a)$ This rule annotates the methods of the monitored classes. The annotation is split up in a pre_set, post_set and an exc_set part.
The pre_set part represents the event entry(Method m) and contains the JML annotation for all of the transitions of the PA for which the event is entry(Method m).
Similarly the post_set and exc_set part represent the event exit(Method m) and excp(Method m) and contains the annotations for each of the transitions of PA containing that event.
The way the annotation is generated for the pre_set, the post_set and the exc_set part is uniformly defined by rule $\alpha_{2,\mathcal{E}}(e,c,mn,a)$

$\alpha_{2,\mathcal{E}}(e,c,mn,a)$ This rule uniformly generates the set annotation for the pre_set, the post_set and the exc_set parts for each monitored method. It does so by selecting all the transitions sharing the same event with method equal to m and the event type equal to entry, exit or exc. Next, for the set of selected transitions $\alpha_{2,\mathcal{T}}(ts,c,a)$ is called.

$\alpha_{2,\mathcal{T}}(ts, c, a)$ This rule generates a CaseSet for the PA control points. When an event is raised, it selects the CP of its current control point via the CaseSet. Next, it selects a transition from the current control point via a nested CaseSet. The nested CaseSet is generated by $\alpha_{2,\mathcal{S}}(t, q, c, a)$.

$\alpha_{2,\mathcal{S}}(t, q, c, a)$ The last rule generates the nested CaseSet for the set of transitions sharing the same event e and control point q. A transition in the CaseSet is selected when the guard and classVar correspond to the object generating the event. Furthermore, the actions of the transition are generated.

$$
\begin{array}{rcl}
\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} \notin a.\mathsf{clnames}\ \mathsf{then}\ c \\
 & & \mathsf{else}\ c\ \texttt{(\#}\quad \mathsf{ghost\_vars} := c.\mathsf{ghost\_vars} \cup a.\mathsf{pa\_var\_decl} \\
 & & \qquad\qquad \mathsf{inv} := \mathsf{And}(\mathsf{Not}(\mathsf{Eq}(\mathsf{cp}, \texttt{halted})), c.\mathsf{inv}) \\
 & & \qquad\qquad \mathsf{methods} := \{\alpha_{2,\mathcal{M}}(c, m, a) \mid m \in c.\mathsf{methods}\} \cup \alpha_{2,\mathcal{R}}(a)\ \texttt{\#)} \\
\alpha_{2,\mathcal{R}}(a) & = & \mathit{Method}\quad m = \texttt{(\#}\ \mathsf{name} := \mathit{reset}, \mathsf{res\_type} = \mathsf{Void}, \mathsf{body} := \\
 & & \{\mathsf{Assign}(cv.\mathit{classType}, cv.\mathit{varName}, \mathit{null}) \mid cv \in a.\mathit{classVars}\}\ \texttt{\#)} \\
\alpha_{2,\mathcal{M}}(c, m, a) & = & m\ \texttt{(\#}\quad \mathsf{pre\_set} := \quad m.\mathsf{pre\_set}; \alpha_{2,\mathcal{E}}(\mathsf{entry}, c, m.\mathsf{name}, a); \\
 & & \qquad\qquad\qquad \mathsf{Assert}(\mathsf{Not}(\mathsf{Eq}(\mathsf{cp}, \texttt{halted}))), \\
 & & \qquad \mathsf{post\_set} := m.\mathsf{post\_set}; \alpha_{2,\mathcal{E}}(\mathsf{exit}, c, m.\mathsf{name}, a) \\
 & & \qquad \mathsf{exc\_set} := m.\mathsf{exc\_set}; \alpha_{2,\mathcal{E}}(\mathsf{exc\_exit}, c, m.\mathsf{name}, a)\ \texttt{\#)} \\
\alpha_{2,\mathcal{E}}(e, c, mn, a) & = & \alpha_{2,\mathcal{T}}(\{t \mid t \in a.\mathsf{trans} \wedge t.\mathsf{event}.\mathit{classType.name} = \mathit{c.name} \wedge \\
 & & t.\mathsf{event.etype} = \texttt{(\#}\ \mathsf{event} := e, \mathsf{mname} := mn\ \texttt{\#)}\}, c, a) \\
\alpha_{2,\mathcal{T}}(ts, c, a) & = & \mathsf{CaseSet}(\{(\mathsf{Eq}(\mathsf{cp}, \mathsf{q}), \alpha_{2,\mathcal{S}}(ts, q, c, a)) \mid \mathsf{q} \in a.\mathsf{cps}\}) \\
\alpha_{2,\mathcal{S}}(t, q, c, a) & = & \mathsf{CaseSet}(\{(t.\mathsf{guard} \wedge (\mathsf{Eq}(Var(t.\mathit{event.Classtype}, t.\mathit{event.classVar}), "") \vee \\
 & & \mathsf{Eq}(Var(t.\mathit{event.Classtype}, t.\mathit{event.classVar}), \mathit{this})); \\
 & & \mathsf{Set}(\mathsf{cp}, t.\mathsf{dest}; \mathit{action}; \mathsf{Assign}(t.\mathit{event.Classtype}, t.\mathit{event.classVar}, \mathit{this})) \mid \\
 & & t \in ts \wedge t.\mathsf{source} = \mathsf{q}\})
\end{array}
$$

Figure 6.3: Translation method: Stage 2

The set annotation generation method is extended in the following way to cope with multiple objects. First, the class variables introduced in the Property Automaton definition are encoded as static class variables. Second, a reset method is generated for each class, which resets its own class variables if necessary. Third, the guard of each transition additionally checks if the static class variable matches the class type of the object currently executing the annotated method. Last, an action is added to each transition which sets the static class variable to the object currently executing the annotated method. This concludes the second stage of the translation method. The next section will illustrate the second stage by a textual example.

6.4 Stage 2: transform example of 4.2

The second stage of the translation method annotates the Java source code with the completed automaton shown in Figure 6.1. First, the translation method enumerates all the automaton states as a integer ghost variables. In this example these are the states s0, s1, s2 and the halted state. Furthermore, the current state of the automaton is represented by the CP integer. The CP variable is

$$
\begin{array}{lll}
\alpha_{3,\mathcal{M}}(P,m) = m(\# & \multicolumn{2}{l}{\mathsf{pre\_set} := \mathsf{Skip}, \mathsf{post\_set} := \mathsf{Skip}, \mathsf{exc\_set} := \mathsf{Skip},} \\
 & \multicolumn{2}{l}{\mathsf{lvars} := \{\mathsf{result}\} \cup m.\mathsf{lvars}, \mathsf{res} := \mathsf{lookup}(\mathsf{result}),} \\
 & \mathsf{body} := & \mathsf{TryCatchFinally}(\\
 & & \quad \mathsf{TryCatchFinally}(\; m.\mathsf{pre\_set}; m.\mathsf{body}; \\
 & & \qquad\qquad\qquad\qquad \mathsf{Assign}(\mathsf{result}, m.\mathsf{res}); m.\mathsf{post\_set} \\
 & & \quad \mathsf{Throwable}, m.\mathsf{exc\_set}, \mathsf{Skip}), \\
 & & \mathsf{RunTimeException}, m.\mathsf{exc\_set}, \mathsf{Skip})\ \#)
\end{array}
$$

Figure 6.4: Translation method: stage 3

adapted accordingly when transitioning to another state.
Secondly, it translates the Terminal class variables $t1, t2$ and $t3$ as static Terminal variables. As explained in the previous section, these class variables are basically references to the terminal objects initiating the transitions in the completed PA.

The translation method creates a CaseSet structure for the pre_set construct when entering the method, for the post_set construct when exiting the method and for the exc_set construct when an exception is thrown by the method. These constructs represent the occasions when a pay method is entered, exited or throws an exception. For example, when the program enters the pay method, the outer CaseSet first checks the current CP.

The inner CaseSet evaluates the guards of transitions starting in the same CP. Because the JML annotations are generated from the completed automaton, there should always be a matching transition.
Next, the transition, for which the guard has evaluated to true, is selected. The JML annotations update the current control point and also set the corresponding class variable.
In case the the current control point evaluates into the trap state, any future transition will also result into a trap state.
The next stage describes the inlining of the JML annotations.

6.5 Stage 3: inlining the set annotations

The third stage inlines the JML set annotations generated in stages into the method body. As shown in Figure 6.4, $\alpha_{3,\mathcal{M}}(P,m)$ moves the pre_set, post_set and exc_set construct into a double TryCatchFinally clause. The inner TryCatchFinally clause first evaluates the pre CaseSet upon entry of a method, then evaluates the body of the method, then evaluates the result of the method and finally evaluates the post caseset before returning the result of the method body.

Furthermore, if the outer TryCatchFinally clause catches any RuntimeExceptions generated by the inner TryCatchFinally clause, it will evaluate the exc CaseSet annotation. If the exc CaseSet is not violated, the thrown exception does not violate the security policy. Otherwise, if a JMLException is thrown,

```
class Terminal {
boolean success = false; /* indicates success of pin transaction

// CPs, Class Variables and current CP
//@ ghost int s0 = 0, s1 = 1, s2 = 2, halted = 3,  cp = s0;
//@ ghost static Terminal t1 = null, t2 = null, t3 = null;
//@ public invariant cp != halted;


/*@  CASESETS
/* pre caseset: does nothing */
pre_set CaseSet [(cp == s0, CaseSet [true , cp = s0)]),
                     (cp == s1, CaseSet [true , cp = s1)]),
                     (cp == s2, CaseSet [true , cp = s2)]),
                     (cp == halted, CaseSet [true , cp = halted)])
                        ]; Assert cp != halted;

/* post caseset: PA Transitions */
post_set  CaseSet [
(cp == s0,
     CaseSet [(success=false;  (Terminal.t1=="" | Terminal.t1==this ), /*guard*/
                     cp = s1; Terminal.t1==this; ),       /*action*/
              (success=true; (Terminal.t1="" | Terminal.t1=this ),  /*guard*/
                     cp = s0; Terminal.t1=this; reset(); ), /*action*/
(cp == s1,
     CaseSet [(success=false; (Terminal.t2="" | Terminal.t2=this ), /*guard*/
                     cp = s2; Terminal.t2=this ),        /*action*/
              (success=true; (Terminal.t2="" | Terminal.t2=this ),  /*guard*/
                     cp = s0; Terminal.t2=this; reset(); ), /*action*/
(cp == s2,
     CaseSet [(success=false; (Terminal.t3="" | Terminal.t3=this ), /*guard*/
                     cp = halted; Terminal.t3=this; ),     /*action*/
            (success=true; (Terminal.t3=null | Terminal.t3=this),  /*guard*/
                     cp = s0; Terminal.t3=this; reset; ),   /*action*/
 (cp == halted,
      CaseSet [true; cp =halted])  ]

/* exc caseset: does nothing */
exc_set  CaseSet [(cp == s0, CaseSet [true , cp = s0)]),
                     (cp == s1, CaseSet [true , cp = s1)]),
                     (cp == s2, CaseSet [true , cp = s2)]),
                     (cp == halted, CaseSet [true , cp = halted)])   ];
@*/
void pay(){/* body pay */}

void reset(){
Terminal.t1 = null; Terminal.t2 = null; Terminal.t3 = null;
}
```

Figure 6.5: Second transformation of example 1

because the Java program violates the security policy.

Because this third stage only wraps the existing CaseSet constructs in TryCatch into FinallyClauses, the definition in Figure 6.4 is the same as defined by Huisman and Tamalet.
An example of the inlined JML annotations of example 1 are shown in figures 6.6 and 6.7.

Compared to Figure 6.5, the third stage altered the annotated source code in the following way:
1. The body of the pay method is surrounded by two TryCatchFinally clauses. The inner TryCatchFinally clause catches any violation of the security policy under normal circumstances. On the other hand, the outer TryCatchFinally clause catches any violation of the security policy under exceptional circumstances. 2. The set annotations at the beginning of the pay method is moved inside the method body.
The preset annotation is moved to the beginning of the second TryCatchFinally clause. Next, the postset annotation is moved to the end of the second TryCatchFinally clause. Last, the excset annotations is duplicated into the catch part of the first and second TryCatchFinally clauses.

The three stages together transforms a security policy, from a PA into JML annotation in the source code of the Java program. As indicated in the introduction, the Mobius Verification Environment can be used to verify the JML annotations statically or dynamically. This concludes the overview of the translation method.

```
class Terminal {
boolean success = false; /* indicates success of pin transaction

// CPs, Class Variables and current CP
//@ ghost int s0 = 0, s1 = 1, s2 = 2, halted = 3,  cp = s0;
//@ ghost static Terminal t1 = null, t2 = null, t3 = null;
//@ public invariant cp != halted;


void pay(){
try{
  try{
  /* pre caseset: does nothing */
  /@*
  pre_set CaseSet [(cp == s0, CaseSet [true , cp = s0)]),
                   (cp == s1, CaseSet [true , cp = s1)]),
                   (cp == s2, CaseSet [true , cp = s2)]),
                   (cp == halted, CaseSet [true , cp = halted)])
                     ]; Assert cp != halted;
  @*/

  //Method body

  /*@
  /* post caseset: PA Transitions */
  post_set  CaseSet [
  (cp == s0,
     CaseSet [(success=false;  (Terminal.t1=="" | Terminal.t1==this ), /*guard*/
                    cp = s1; Terminal.t1==this; ),       /*action*/
             (success=true; (Terminal.t1="" | Terminal.t1=this ),  /*guard*/
                    cp = s0; Terminal.t1=this; reset(); ), /*action*/
  (cp == s1,
     CaseSet [(success=false; (Terminal.t2="" | Terminal.t2=this ), /*guard*/
                    cp = s2; Terminal.t2=this ),        /*action*/
             (success=true; (Terminal.t2="" | Terminal.t2=this ),  /*guard*/
                    cp = s0; Terminal.t2=this; reset(); ), /*action*/
  (cp == s2,
     CaseSet [(success=false; (Terminal.t3="" | Terminal.t3=this ), /*guard*/
                    cp = halted; Terminal.t3=this; ),     /*action*/
           (success=true; (Terminal.t3=null | Terminal.t3=this),  /*guard*/
                    cp = s0; Terminal.t3=this; reset; ),   /*action*/
   (cp == halted,
      CaseSet [true; cp =halted])  ]
  @*/

  return null;//no result
```

Figure 6.6: Third transformation of example 1: Part 1

```
  }catch(Throwable e1) {
  /* exc caseset: does nothing */
  exc_set  CaseSet [(cp == s0, CaseSet [true , cp = s0)]),
                    (cp == s1, CaseSet [true , cp = s1)]),
                    (cp == s2, CaseSet [true , cp = s2)]),
                    (cp == halted, CaseSet [true , cp = halted)])   ];
  @*/
  } finally{ /* do nothing */ }
} catch(RunTimeException e2) {
   //Repeat Exc Caseset
} finally{ /* do nothing */ }
}
}

void reset(){
Terminal.t1 = null; Terminal.t2 = null; Terminal.t3 = null;
}
```

Figure 6.7: Third transformation of example 1: part 2

Chapter 7

Conclusion

The conclusion of this assignment revisits the research goals given in Section 1.3. The main goal was to investigate how the introduction of objects in the definition of the Property Automaton (PA) altered the existing translation method of Huisman and Tamalet [7].
In order to formulate an answer to the main research goal the definitions of the syntax, Security Automaton, Property Automaton and operational semantics have been reviewed and will be summarized in the next paragraph.

7.1 Adaptation to the Java syntax

The simplified Java language of Huisman and Tamalet only supported the monitoring of static classes. As opposed to objects, static classes only exists once in a Java program and therefore cannot be instantiated. In this assignment, the introduction of objects alleviated the restriction of static classes and therefore enabled the monitoring of a set of objects instances. For simplicity, the newly defined simplified Java language only supports the monitoring of object instances and not of static classes. Furthermore, the introduction of objects in the syntax of chapter 2 mainly affected the expression and method definition, because the assignment and the lookup of a variable can refer to an object variable. Last, it introduced the concept of static and non-static variables, but it did not include static methods and classes.

7.2 Adaptation to the security and property automata

A security policy, in relation to a program, describes an execution, which is unacceptable for one reason or another. Security automata are a class of Büchi automata that accept safety properties and are similar to ordinary non-deterministic finite state automata [11].

The PA can be thought of as a Finite State Machine (FSM). However, the transitions in a PA are generally more complex.

The Security Automata definition in chapter 3 is similar to the Property Automaton definition and have the following in common:

- Both the SA and PA have a set of control points, which represent the program state.
- Both the SA and PA have a set of initial control points in which the monitored program starts.
- Both for the SA and PA, the transition function solely depends on the current state and the transition
- The transitions definition of the PA is more complex than the transition definition of SA. The transition of a SA only consists of a single action, while the transition definition of the PA depends on a guard, a class variable and results into an action.

The introduction of objects in the Java syntax definition resulted into the introduction of class variables in the PA definition. A class variable has a name and a classtype. The class variable is associated with a single object instance at run-time and limits the execution of transitions with the same class variable to the same object instance. I.e. if there are three consecutive transitions with each containing a class variable, the transitions having the same class variable should be executed by the same object.

7.3 Adaptation to the operation semantics of the Java language

The introduction of objects did affect the operational semantics of variable lookup, assignment evaluation and class initialization in comparison to the one given in Huisman and Tamalet [7].
The variable lookup had to deal with the introduction of static and non static variables. It did so by introducing an object reference, which in combination with the variable name, could point to a field variable in an object or a local variable.
The assignment evaluation definition is affected in a similar way. It also made use of a reference object. But instead of pointing to a class field or local variable for retrieval, the resolved referece is used to store a value.
Last, the Java language supports the creation of objects. All the objects with the same class type have their own non static variable store. However the static variables of the same classtype are shared among all of the object. Thus when the object of a certain classtype is initialized for the first time, both the static and object class store should be created. However, for succeeding objects of the same classtype only object store needs to be initialized.

7.4 Adaptations to the translation method

The translation method of Huisman and Tamalet [7] translates a Property Automaton into JML annotations of the monitored program. The introduction of objects enabled the translation method to monitor multiple objects at the same time. This resulted also in extending the annotation generation method to annotate multiple classes based on a single Property Automaton.

The biggest adaptation was the introduction of class variables into the JML annotations. As explained earlier, the class variables restrict the set of objects allowed to generate a transition event. An object is only allowed to generate a transition if either the class variable is not yet associated with an object or if the object matches the reference of the class variable.

This all together results in that a single automaton can monitor multiple object instances. Class variables can be used to force transitions to be executed by the same object. The one-to-many relation between automaton and the object instance has as disadvantage, that the interleaving of execution of method of different object results in a state explosion [6].

7.5 Future work

In order to counter the state explosion problem, one could relax the constraint of having multiple automatons monitoring multiple objects. When a automaton is parametrized, one template automaton can be used to create a automaton instance monitoring a single object. I.e. a template automaton describing a security policy for a buffer, can be duplicated for each buffer object.
Another possible future research topic is the extension of the simplified Java language. The simplified Java language in this assignment is only extended with objects, but the introduction of multi threading will also improve the universe of definable property automatons.
Finally, the original and the extended translation method is only formalized on paper and not yet implemented as a tool.

Bibliography

[1] B. Alpern and F.B. Schneider, Recognizing safety and liveness, Distributed computing **2** (1987), no. 3, 117–126.

[2] G. Barthe, Mobius, securing the next generation of Java-based global computers, ERCIM News (2005).

[3] J.L. Dalley, The art of software testing, Aerospace and Electronics Conference, 1991. NAECON 1991., Proceedings of the IEEE 1991 National, IEEE, pp. 757–760.

[4] R. Di Pietro and L.V. Mancini, Security and privacy issues of handheld and wearable wireless devices, Communications of the ACM **46** (2003), no. 9, 74–79.

[5] J. Gosling, The Java language specification, Prentice Hall, 2000.

[6] H. Hermanns, B. Haverkort, and J.P. Katoen, On the use of model checking techniques for dependability evaluation, System (2000).

[7] M. Huisman and A. Tamalet, A formal connection between security automata and JML annotations, Fundamental Approaches to Software Engineering (2009), 340–354.

[8] G.T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. Cok, P. Muller, J. Kiniry, P. Chalin, and D.M. Zimmerman, JML reference manual, Department of Computer Science, Iowa State University (2006).

[9] J. Pan, Software testing, 1999, This is an electronic document. Date of publication: Spring, 2011. Date retrieved: July 10, 2011.

[10] M. Pavlova, G. Barthe, L. Burdy, M. Huisman, and J.L. Lanet, Enforcing high-level security properties for applets, Smart Card Research and Advanced Applications VI (2004), 1–16.

[11] F.B. Schneider, Enforceable security policies, ACM Transactions on Information and System Security (TISSEC) **3** (2000), no. 1, 30–50.

[12] TRUSTe, Truste privacy seals & services, 2011, This is an electronic document. Date of publication: May 19, 2011. Date retrieved: July 10, 2011.

[13] D. von Oheimb, Analyzing Java in isabelle/HOL: Formalization, type safety and hoare logic, Ph.D. thesis, PhD thesis, Technische Universitat Munchen, 2001.